Banner Oracle
PL/SQL and Database Objects Training Workbook

January 2007

Using Oracle for Banner 7

SUNGARD HIGHER EDUCATION

What can we help you achieve?

Prepared By: SunGard Higher Education
4 Country View Road
Malvern, Pennsylvania 19355
United States of America

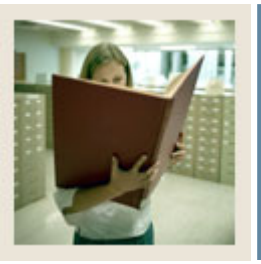


Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

Table of Contents (Continued)

Table of Contents (Continued)

Table of Contents (Continued)

Section A: Introduction

Lesson: Overview

Jump to TOC

Workbook goal

SQL (structured query language) is the core language used to interact with an Oracle database. PL/SQL, or the procedural language of SQL, can be considered the second layer of language because it allows the use of conditional statements.

Introduction to Oracle and SQL acquainted you with the basics of SQL. This course will expand upon this base by working introducing the procedural concepts of SQL including stored procedures, functions, packages, triggers, and code optimization.

Participants in this course will be able to

- write statements to obtain information from the database
- write statements to generate reports
- manipulate data and process transactions
- write programs in which SQL statements are enclosed within procedural statements (such as IF statements)
- create program units, such as procedures and functions, which can be stored in the database.

Intended audience

PL/SQL is used for all types of database activities by many types of users. However, in order for attendees to receive the optimum benefit of this training, Sungard Higher Education recommends that prospective students come from one of the following groups:

- System administrators
- Database administrators
- Security administrators
- Application programmers
- Decision support system personnel
- End users who have extensive contact with the database

Prerequisites

To complete this workbook, you should have:

- completed the Education Practices computer-based training (CBT) tutorial "Banner 7 Fundamentals," or have equivalent experience navigating in the Banner system
- completed the Introduction to Oracle training workbook

Section B: PL/SQL Basic Structure

Lesson: Overview

Jump to TOC

Introduction

This section provides an overview of the basic concepts of PL/SQL.

Objectives

This section will examine the following:

- The basic structure of PL/SQL
- How PL/SQL interprets and executes statements

Section contents

Section B: PL/SQL Basic Structure

Lesson: PL/SQL Overview

Jump to TOC

What is PL/SQL?

PL/SQL is Oracle's procedural extension to industry-standard SQL. With PL/SQL, you can use SQL statements to manipulate Oracle data and flow-of-control statements to process the data. You can also declare constants and variables, define procedures and functions, and trap runtime errors. Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages

Example

PL/SQL allows you to enclose your SQL statements with conditions.

```
  /*
  This procedure will either insert or delete a record from the SWRIDEN
  table, depending on the parameter of
  action_in.
  */
  CREATE OR REPLACE PROCEDURE maintain_pidms
  (pi_action     IN VARCHAR2,
   pi_pidm       IN NUMBER :=NULL,
   pi_id         IN VARCHAR2,
   pi_last_name  IN VARCHAR2 := NULL,
   pi_first_name IN VARCHAR2 := NULL,
   pi_mi         IN VARCHAR2 := NULL)
  IS
  BEGIN
     IF pi_action = 'DELETE' THEN
        DELETE FROM swriden
              WHERE swriden_pidm = pi_pidm;
     ELSIF pi_action = 'INSERT' THEN
        INSERT INTO swriden
              (swriden_pidm, swriden_id, swriden_last_name,
               swriden_first_name, swriden_mi,
               swriden_activity_date
              )
            VALUES (pi_pidm, pi_id, pi_last_name,
                    pi_first_name, pi_mi, SYSDATE
                   );
     END IF;
  END;
/
```

Section B: PL/SQL Basic Structure

Lesson: PL/SQL Block Structure

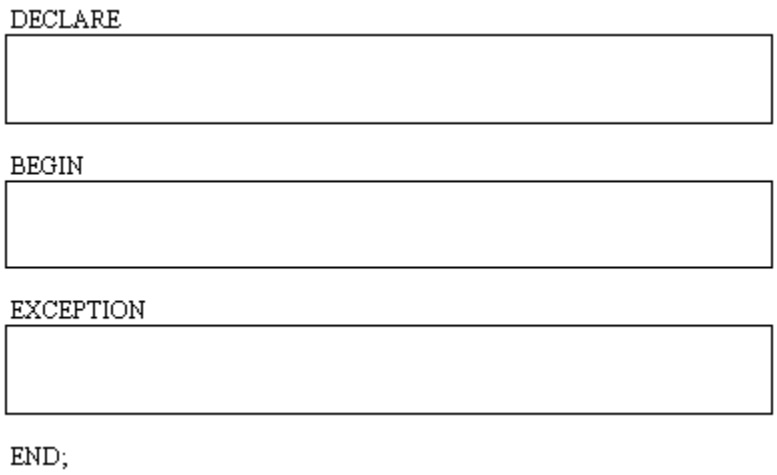
Jump to TOC

Block structure

PL/SQL is a *block-structured* language. That is, the basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or subproblem to be solved A block (or sub-block) lets you group logically related declarations and statements

A PL/SQL block has three parts: a declarative part, an executable part, and an exception-handling part. Only the executable part is required

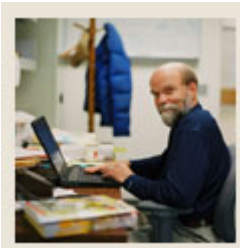
Diagram



Named vs Anonymous

There are named and anonymous PL/SQL blocks. A named block is a function, procedure, package, or trigger that is given a formal name and the associated PL/SQL code is stored in the database for use by other SQL and PL/SQL processes. These are sometimes referred to as 'stored' PL/SQL

An anonymous block is normally either written ad-hoc or stored in an SQL file for execution via running the SQL file. It cannot be called by SQL commands or other stored (named) PL/SQL blocks.



Header

For named blocks only, such as procedures, functions, and packages, the Header section assigns a label to a given block and specifies the type of block to be defined.

Declaration

The declaration section is the part of the block that declares variables, cursors, and sub-blocks that are referenced in the execution and exception sections.

Execution

This is the part of the PL/SQL block containing the executable statements which can include standard SQL statements as well as procedural or looping statements.

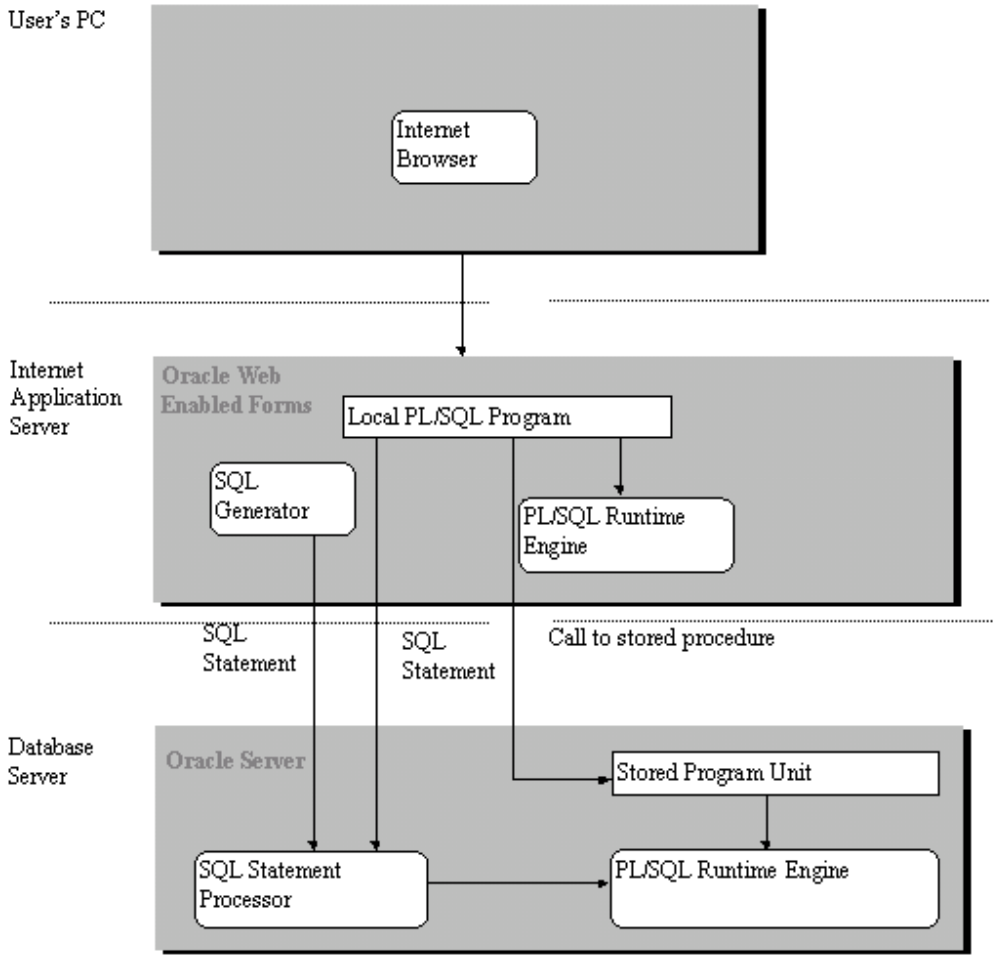
Exception

The section handles exceptions to normal processing (warnings and error conditions).

Notes

- Blocks can contain sub-blocks
- Executable statements end with a semi-colon
- Declared objects exist within a certain scope (discussed in the following section)

Diagram


User's PC
Internet Browser
Internet Application Server
Oracle Web Enabled Forms
Local PL/SQL Program
SQL Generator
PL/SQL Runtime Engine
SQL Statement
SQL Statement
Call to stored procedure
Database Server
Oracle Server
Stored Program Unit
SQL Statement Processor
PL/SQL Runtime Engine

Section B: PL/SQL Basic Structure

Lesson: Conventions

Jump to TOC

Case Restrictions

As with SQL, there are no case restrictions when writing PL/SQL. Code can be written in upper, lower, or mixed. However, common conventions have key words or commands capitalized with other statements or variables in lower case.

When referring to any variable or object in PL/SQL they are case insensitive. Even if a variable is defined in upper case it can be referred to later in lower or mixed case.

```
DECLARE
      lv_my_numb       number(10,2);
BEGIN
      SELECT count(*) INTO lv_my_numb
        FROM spriden;
END;
```

Naming Standards

It is a good idea to establish naming standards early in PL/SQL development. This makes the code easier to read and maintain, especially if the author is not the person making changes to an existing program.

Common naming standards include prefix assignments for parameters and variables as well as names for stored PL/SQL objects. Common prefixes for stored objects include PKG_ for stored packages, SP_ or P_ for stored procedures, and F_ for stored functions.

In this course we will strive to maintain the following conventions:

Parameters:

pi_	Parameter IN
po_	Parameter OUT
pio_	Parameter IN/OUT

Variables:

lv_	Locally declared variable
gv_	Globally declared variable
vi_	Parameter declared as incoming and assigned to a local variable
vo_	Parameter declared as outgoing and assigned to a local variable
vio_	Parameter declared as in or out and assigned to a local variable

Section B: PL/SQL Basic Structure

Lesson: Conventions (continued)

Jump to TOC

Formatting

It is a good practice to get into the habit of formatting your PL/SQL from the start. Formatting, like naming conventions, increases readability and makes code easier to maintain.

Line up the main sections of the code and indent statements within (3-5 spaces is common). For loops or sub-blocks, continue to indent. Lining up your SQL statements also improves readability.

```
    DECLARE
        lv_count   number(2);
    BEGIN
        SELECT ……
          INTO ……
          FROM ……
         WHERE ……

        BEGIN
            LOOP
                lv_count := lv_count + 1;
                EXIT when lv_count > 10;
            END LOOP;
        END;
    EXCEPTION
         WHEN no_data_found THEN

    END;
/
```

Execute PL/SQL

To execute your anonymous PL/SQL block in SQL*Plus you must put a forward slash (/) on the line after the last END; statement.

```
SQL>  DECLARE
  2    lv_my_numb  number(10,2);
  3   BEGIN
  4    SELECT count(*) INTO lv_my_numb
  5      FROM swriden;
  6   END;
  7   /
PL/SQL procedure successfully completed.

SQL>
```

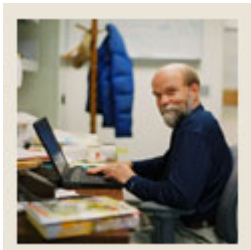
If you make a typographical error or syntactical error in your PL/SQL you must still execute the block to be able to go back and make corrections.

Errors

Once you receive an error for syntactical or typographical errors, you can correct your PL/SQL through the line editor or by using the EDIT command in SQL*Plus.

```
 SQL>  DECLARE
  2    lv_my_numb  number(10,2);
  3   BEGIN
  4    SELECT count(*) INTO lv_my_numb
  5      FROM swridez;
  6*   END;
SQL> /
    FROM swridez;
         *
ERROR at line 5:
ORA-06550: line 5, column 10:
PL/SQL: ORA-00942: table or view does not exist
ORA-06550: line 4, column 3:
PL/SQL: SQL Statement ignored
```

NOTE: Oracle will attempt to point out where the error occurred but if it the exact location is not obvious to the PL/SQL compiler it may give an ambiguous error.



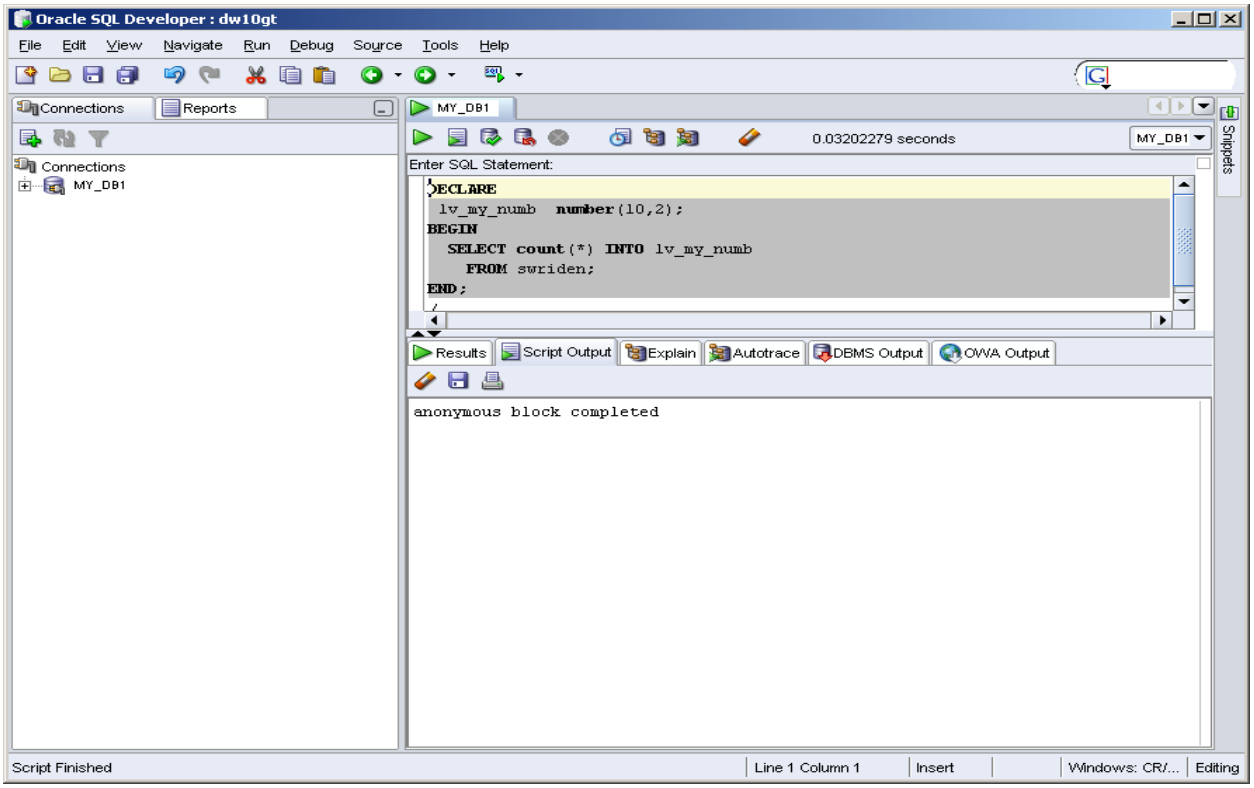
Section B: PL/SQL Basic Structure

Lesson: Running Anonymous PL/SQL (continued)

Jump to TOC

Running PL/SQL from SQL Developer

SQL Developer is a new tool from Oracle that will one day replace SQL*Plus. While most of this course will focus on using SQL*Plus you may also use SQL Developer. Differences are pointed out where appropriate.



 Use this button to run a standard SQL statement. The results will appear in the Results tab

 Use this button to run PL/SQL. The results will appear in the Script Output tab

 This button will commit the current transaction.

 This button will roll back the current transaction.

 This button allows you to see a history of SQL statements you have issued

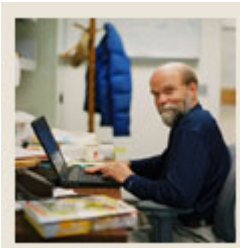
 This button erases the current window whether it is the SQL window or the Script Results window

Section B: PL/SQL Basic Structure

Lesson: PL/SQL Versions/History

Jump to TOC

VERSION / RELEASE	CHARACTERISTICS
Version 1.0 (Oracle 6)	First available in SQL*Plus as a batch processing. PL/SQL was then implemented within SQL*Forms Version 3, the predecessor of Oracle Forms.
Version 1.1	Available only in the Oracle Development tools. Supports client-side packages and allows client-side programs to execute stored code transparently.
Version 2.0 (Oracle 7.0)	Adds support for stored procedures, functions, packages, programmer-defined records, PL/SQL tables, and many package extensions, including DBMS_OUTPUT and DBMS_PIPE.
Version 2.1 (Oracle 7.1)	Supports user-defined subtype, enables stored functions in SQL statements, and offers dynamic SQL with the DBMS_SQL package. DDL statements can now be executed from within PL/SQL.
Version 2.2 (Oracle 7.2)	Implements binary "wrapper" for PL/SQL programs to protect source code, supports cursor variables for embedded PL/SQL environments such as Pro*C, and makes available database-driven job scheduling with DBMS_JOB package.
Version 2.3 (Oracle 7.3)	Enhances functionality of PL/SQL tables, offers improved remote dependency management, adds file I/O capabilities to PL/SQL, and completes the implementation of cursor variables.
Version 8.0 (Oracle 8.0)	Object types and methods were enhanced. Collection types – nested tables and varrays were added. Advanced queuing option, support for External procedures and LOB enhancements.
Version 9.0.1 (Oracle 9.0)	Enhanced integration of SQL and PL/SQL parsers. Added CASE statements and expressions; merge statement; Type Evolution; new Date/Time types; and Native Compilation of PL/SQL code. Improved Globalization and National language Support. Enhanced table functions and Cursor expressions. Allows multilevel collections. Better integration of LOB Datatypes.
Version 9.2 (Oracle 9.2)	Enhanced Insert/Update/Select of entire PL/SQL records,. Added Associative arrays and User-defined constructors. Enhanced UTL_FILE with new functions. Added *treat, is of,* and *only* function for data types.
Version 10.1 (Oracle 10.1)	Performance improvements include better integer performance, reuse of expression values, simplification of branching code, better performance for some library calls, and elimination of unreachable code. Native compilation easier and more integrated; fewer initialization parameters, less compiler configuration, object code stored in the database, and compatibility with Oracle Real Application Clusters. The *FORALL* statement handles associative arrays and nested tables with deleted elements. New functions *SCN_TO_TIMESTAMP* and *TIMESTAMP_TO_SCN* allow you translate between a date/time, and the system change number that represents the database state at a point in time.
Version 10.2 (Oracle 10.2)	Offers Conditional Compilation feature, which enables you to selectively include code depending on the values of the conditions evaluated during compilation.



Section B: PL/SQL Basic Structure

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self check activity.

Exercise 1

What are the three main parts of a PL/SQL block?

Exercise 2

Describe an anonymous block.

Section C: Declaring Variables

Lesson: Overview

Jump to TOC

Introduction

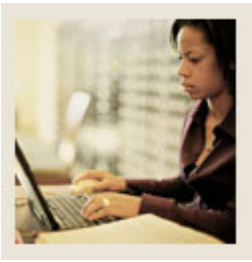
Communication with the database takes place through variables in the PL/SQL block. Variables are memory locations, which can store data values. As the program runs, the contents of variables can and do change. Information for the database can be assigned to a variable, or the contents of a variable can be inserted into the database. Every variable has a specific type as well, which describes what kind of information and be stored in it.

Objectives

This section will examine the following:

- Legal vs. Illegal variables
- Understanding the Scope and visibility of variables

Section contents



Section C: Declaring Variables

Lesson: Declaring Variables

Jump to TOC

Syntax

```
Identifier [CONSTANT] datatype [NOT NULL] [:= plsql_expression];
```

Examples

Number

```
lv_pidm             NUMBER;
lv_amount           NUMBER(7,2);
lv_tax              CONSTANT DECIMAL := .06;
lv_fee              NUMBER(7,2) := 0;
```

CHAR/ VARCHAR2

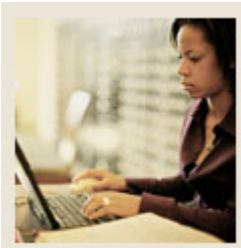
```
lv_last_name        VARCHAR2(30);
lv_institution      VARCHAR2(20)  NOT NULL := 'ABC University';
```

DATE

```
lv_application_date DATE;
lv_today            DATE := SYSDATE;
lv_bill_date        DATE := TO_DATE('10/10/2006','MM/DD/YYYY');
```

BOOLEAN

```
lv_amount_paid      BOOLEAN;
lv_registered       BOOLEAN  NOT NULL := FALSE;
```



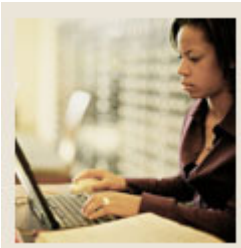
Section C: Declaring Variables

Lesson: Built-in Datatypes

Jump to TOC

Scalar Types

CATEGORY	DATATYPE
Number	BINARY_INTEGER
	DEC
	DECIMAL
	DOUBLE PRECISION
	FLOAT
	INT
	INTEGER
	NATURAL
	NATURALN
	NUMBER
	NUMERIC
	PLS_INTEGER
	POSITIVE
	POSITIVEN
	REAL
	SIGNTYPE
	SMALLINT
Character	CHAR
	CHARACTER
	LONG
	LONG_RAW
	NCHAR
	NVARCHAR2
	RAW
	VARCHAR (for backward compatibility only)
	VARCHAR2
	MLSLABEL
Boolean	BOOLEAN
Date-time	DATE
	INTERVAL DAY TO SECOND
	INTERVAL YEAR TO MONTH
	TIMESTAMP
	TIMESTAMP WITH LOCAL TIME ZONE
	TIMESTAMP WITH TIME ZONE



Composite Types

CATEGORY	DATATYPE
	RECORD
	TABLE
	VARRAY

Reference Types

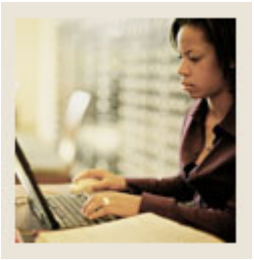
CATEGORY	DATATYPE
	REF CURSOR
	REF object_type

LOB Types

CATEGORY	DATATYPE
	BFILE
	BLOB
	CLOB
	NCLOB

Special Types

CATEGORY	DATATYPE
	ROWID
	UROWID



Association

You can associate PL/SQL objects with certain attributes from another object. It can either be another declared variable or a database item.

%TYPE

The variable takes on the data type of the referenced variable or database column.

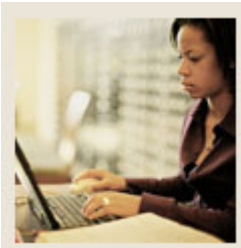
```
DECLARE
   lv_amount         NUMBER(7,2);
   lv_balance        twraccd.twraccd_amount%TYPE;
   lv_internal_id    swriden.swriden_pidm%TYPE;
```

%ROWTYPE

ROWTYPE creates a record type with a field for each column in the specified table.

```
DECLARE
   lv_swriden_row    swriden%ROWTYPE;
```

When creating variables that are to store values selected from tables it is good practice to use the %TYPE or %ROWTYPE assignment so that if the column or table changes in the database, the variable in the PL/SQL program unit also takes on that change. Without this assignment, each PL/SQL program unit referencing that table or column would need to be examined to determine if the change to the table affected how that code processes the data and if the code needed to be changed to reflect the table change.



Section C: Declaring Variables

Lesson: Scoping Rules

Jump to TOC

Resolving references

References to a variable are resolved according to its scope and visibility.

Scope

The region of a program unit from which you can reference the identifier.

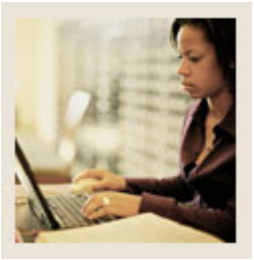
Visibility

An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.

Rules

Identifiers declared in a PL/SQL block are considered local to that block and global to all its sub-blocks.

If a global identifier is redefined in a sub-block, then both identifiers remain in scope. Within the sub-block, however, only the local identifier is visible because you must use a qualified name to reference the global identifier.

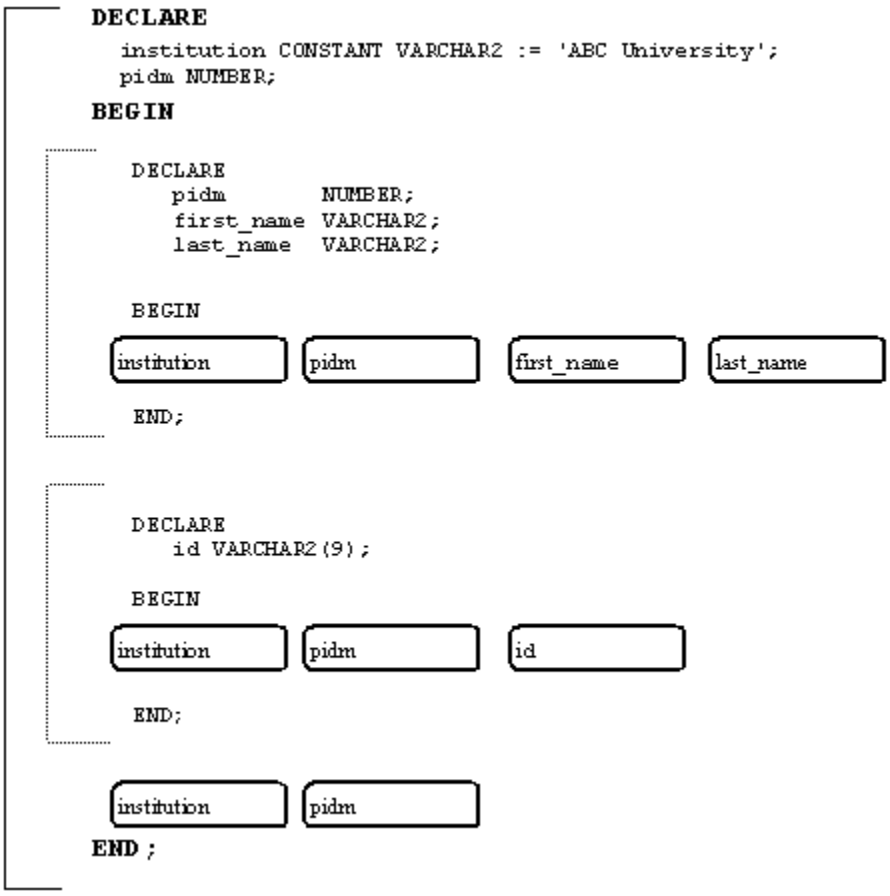


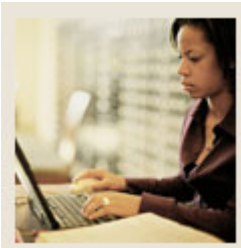
Section C: Declaring Variables

Lesson: Scoping Rules (Continued)

Jump to TOC

Example





Section C: Declaring Variables

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

How can you create a variable that has the same characteristics as a column in the database?

Exercise 2

Mark the following as legal or illegal definitions:

```
lv_Pidm  NUMBER;                    (Legal or Illegal?)
lv_Pidm  NUMBER(8) :=  null;        (Legal or Illegal?)
lv_PIDM  NUMBER(8) := 0;            (Legal or Illegal?)
lv_PIDM  NUMBER(8) NOT NULL;        (Legal or Illegal?)
lv_name  varchar2;                  (Legal or Illegal?)
lv_state char;                      (Legal or Illegal?)
lv_today date := '01/01/2003';      (Legal or Illegal?)
lv_state varchar2(4) := 'FLORIDA';  (Legal or Illegal?)
update   varchar2(7) := 'FLORIDA';  (Legal or Illegal?)
```

Section D: SQL Statements within PL/SQL

Lesson: Overview

Jump to TOC

Introduction

SQL statements within PL/SQL allow us to manipulate data in the database.

Objectives

Upon completion of this section, each attendee will be able to

- use SQL statements within PL/SQL
- identify SQL functions supported by PL/SQL

Section contents

Section D: SQL Statements within PL/SQL

Lesson: Comments

Jump to TOC

Comments

Comments can be added to any PL/SQL code to help explain the actions being undertaken. It is a good idea to put in many comments throughout your program to provide documentation and a reference if some part of the code needs to be changed at a later date.

A single line can be commented using two dashes (--) at the start of the line.

Entire sections of code or multi-line comments can be achieved through block comment characters /* to start a block of comments and */ to end a block of comments.

```
BEGIN
--  This section of code is for….
          SELECT ……
             FROM…..

/*  This section of code actually manipulates data
    It was put in place by John Q. Codewriter
    To accommodate changes made for Banner upgrade to version 7.3
*/
          UPDATE …….
                SET ……
           WHERE ……;
END;
/
```

Section D: SQL Statements within PL/SQL

Lesson: Data Manipulation

Jump to TOC

Versions

All PL/SQL versions support SQL data manipulation allowing you to modify data in the database. These statements include INSERT, UPDATE, and DELETE which are often referred to as DML statements.

INSERT

```
/*
Inserts a record into the swbpers table
*/
DECLARE
   pi_pidm          NUMBER(8) := 1021;
   pi_ssn           VARCHAR2(9) := '987654321';
   pi_birth_date    DATE  := TO_DATE('10-FEB-1973','DD-MON-YYYY');
   pi_mrtl_code     VARCHAR2(1) := 'S';
   pi_sex           VARCHAR2(1) := 'F';
   pi_confid_ind    VARCHAR2(1) := NULL;
BEGIN
   INSERT INTO swbpers
               (swbpers_pidm, swbpers_ssn, swbpers_birth_date,
                swbpers_mrtl_code, swbpers_sex,
                swbpers_confid_ind, swbpers_activity_date
               )
        VALUES (pi_pidm, pi_ssn, pi_birth_date,
                pi_mrtl_code, sex_in,
                pi_confid_ind, SYSDATE
               );
END;
/
```

UPDATE

```
  /*
  Updates the old zip code with the new zip code.
  */
  DECLARE
     lv_zip_old VARCHAR2(10) := '19380';
     lv_zip_new VARCHAR2(10) := '19382';

  BEGIN
     UPDATE swraddr
        SET swraddr_zip = lv_zip_new
      WHERE swraddr_zip = lv_zip_old;
  END;
```

DELETE

```
  /*
  Deletes all rows from the account table where
  the term matches '200601'
  */
  DECLARE
     lv_term VARCHAR2(6) := '200601';
  BEGIN
     DELETE FROM twraccd
           WHERE twraccd_term_code = lv_term;
  END;
/
```

DDL

Until PL/SQL version 2.0, Data Definition Language (DDL) statements were not allowed. This includes creating, altering, and dropping objects from the database. Such statements are made possible through the built-in package DBMS_DDL. They can also be issued through a new convention introduced in Oracle 8i called EXECUTE IMMEDIATE. Both options will be discussed in later sections.

Section D: SQL Statements within PL/SQL

Lesson: Retrieving Data

Jump to TOC

SELECT INTO

A SELECT INTO statement is the only DML that returns data. You need to provide the host variable names in which this data is to be stored, specified in the INTO clause. A variable for each column being selected must be declared in the declaration section.

A SELECT INTO statement must return only one row. If zero or multiple rows are returned, an error condition occurs. If you would like to return multiple rows, use cursors which are discussed later.

Syntax

```
SELECT  col1, col2, ...
  INTO  var1, var2 ...
  FROM  table_name
 WHERE  ...
```

Example

```
/*
Retrieves the last and first name for PIDM 12340 into the host
variables.
*/
DECLARE
   lv_last_name  swriden.swriden_last_name%TYPE;
   lv_first_name swriden.swriden_first_name%TYPE;
BEGIN
   SELECT swriden_last_name, swriden_first_name
     INTO lv_last_name, lv_first_name
     FROM swriden
    WHERE swriden_pidm = 12340
      AND swriden_change_ind IS NULL;

    -- data manipulation

END;
/
```

Section D: SQL Statements within PL/SQL

Lesson: Process Transactions

Jump to TOC

Transactions

A transaction is a set of manipulation statements between COMMITs or saving of changes. Many DML statements can be issued in a PL/SQL block. To segregate each change or section of changes a marker called a SAVEPOINT can be issued. SAVEPOINTS allow certain parts of the transaction to be undone if some error or event occurs.

SAVEPOINT

```
SAVEPOINT < marker name >
```

ROLLBACK TO

```
ROLLBACK [WORK] TO [SAVEPOINT]  < marker name >
```

Sample

```
BEGIN
  INSERT INTO swriden (swriden_pidm, swriden_id, swriden_last_name,
                       Swriden_first_name, swriden_activity_date)
       VALUES (pidm_sequence.NEXTVAL, '123456789',
               'Smith', 'John', SYSDATE);
  SAVEPOINT A;
  . . . .
  SAVEPOINT B;
  . . . .
  ROLLBACK TO SAVEPOINT A;
  COMMIT;
END;
/
```

SAVEPOINT Naming

SAVEPOINTs can have any name – as small as a single character or an entire word or series of words connected by underscores.

A SAVEPOINT name can be re-used but when you re-use a savepoint name it *moves* the savepoint to the new spot and will not allow the transaction to be undone back to the original spot.

Section D: SQL Statements within PL/SQL

Lesson: Self Check

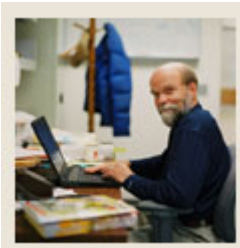
Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

Write a PL/SQL block to insert a row about you into SWRIDEN. The values for the insert must be declared in a declaration section. (Either prompt for the values using the *&variable* convention or hard code the values in the declaration section.) Check to make sure your row was inserted.



Section E: Conditional, Iterative, Sequential Control

Lesson: Overview

Jump to TOC

Introduction

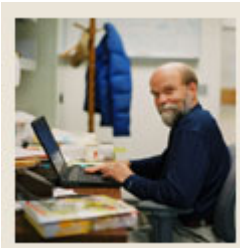
One of the powers of PL/SQL is its ability to provide conditional, iterative and sequential control of statements.

Objectives

Upon completion of this section, each attendee will be able to

- write a PL/SQL procedures to conditionally execute SQL
- identify three uses for PL/SQL statements
- identify four types of loops
- write a PL/SQL procedure using a loop structure.

Section contents



IF-THEN

Use the IF-THEN construct when you want to execute one or more statements if the condition yields TRUE.

```
IF <condition>
THEN
   <TRUE sequence of statements>
END IF;
```

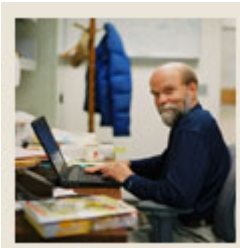
```
IF average_gpa > 3.0 THEN
   student_status := 'HONORS';
END IF;
```

IF-THEN-ELSE

Use the IF-THEN-ELSE statement when you want to choose between two mutually exclusive actions.

```
IF <condition>
THEN
   <TRUE sequence of statements>
ELSE
   <FALSE or NULL sequence of statements>
END IF;
```

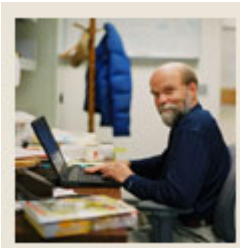
```
IF average >= .70
THEN
   student_status := 'PASSED';
ELSE
   student_status := 'FAILED';
END IF;
```



IF-THEN-ELSIF

Use the IF-THEN-ELSIF statement when you want to select an action from several mutually exclusive alternatives.

```
  IF <condition>
  THEN
     <TRUE sequence of statements>
  ELSIF <condition>
  THEN
     <TRUE sequence of statements>
  ELSE
     <FALSE or NULL sequence of statements>
  END IF;


  /* This block determines whether the institution size is small,
  medium, or large based on the number of records in the SWRIDEN
  table. */

  DECLARE
     lv_num_students     NUMBER(10);
     lv_institution_size VARCHAR2(7);
  BEGIN
     SELECT COUNT(*)
       INTO lv_num_students
       FROM swriden
      WHERE swriden_change_ind IS NULL;
     IF lv_num_students < 5000 THEN
          lv_institution_size := 'Small';
       ELSIF  lv_num_students BETWEEN 5000 AND 14999 THEN
          lv_institution_size := 'Medium';
       ELSE
          lv_institution_size := 'Large';
     END IF;
  END;
/
```



CASE

In Oracle 9i, you can also use the CASE statement when you want to select an action from several mutually exclusive equality tests.

Equality Case

```
CASE variable
WHEN value1 THEN
   <value1 sequence of statements>;
WHEN value2 THEN
   <value2 sequence of statements>;
WHEN value3 THEN
   <value3 sequence of statements>;
. . .
ELSE
   <else sequence of statements>;
END CASE;
```

```
    /* This block determines msg value based on column sex. */

    DECLARE
       lv_sex       SWBPERS.swbpers_sex%TYPE;
       lv_msg       varchar2(10);
    BEGIN
       SELECT swbpers_sex
         INTO lv_SEX
         FROM SWBPERS
        WHERE swbpers_pidm = '12345';
       CASE lv_sex
          WHEN 'M' THEN lv_msg := 'Male';
          WHEN 'F' THEN lv_msg := 'Female';
          ELSE          lv_msg := 'Unknown';
       END CASE;
       . . .
    END;
/
```

Inequality Case

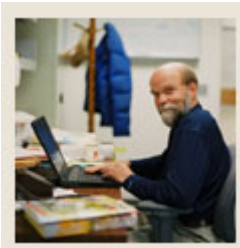
The CASE statement can also be used to test for inequality as well as equality. The syntax is slightly different as the variable does not appear after the CASE statement but instead as part of the WHEN statements.

```
CASE
WHEN variable operator value1(s) THEN
   <value1 sequence of statements>;
WHEN variable operator value2(s) THEN
   <value2 sequence of statements>;
WHEN variable operator value3(s) THEN
   <value3 sequence of statements>;
. . .
ELSE
   <else sequence of statements>;
END CASE;
```

Operator can be options like >, <, >=, <=, <>, LIKE, NOT LIKE, BETWEEN, IN, NOT IN, etc.

Example

```
DECLARE
   lv_count  NUMBER;
   lv_size   VARCHAR2(30);
BEGIN
   SELECT COUNT(*)
      INTO lv_count
      FROM swbpers;
   CASE
      WHEN lv_count < 500 THEN
         lv_size := 'SMALL';
      WHEN lv_count BETWEEN 500 AND 999 THEN
         lv_size := 'MEDIUM';
      WHEN lv_count > 999 THEN
         lv_size := 'LARGE';
      ELSE
         lv_size := 'UNKNOWN';
   END CASE;
   DBMS_OUTPUT.PUT_LINE(lv_size);
END;
```



Section E: Conditional, Iterative, Sequential Control

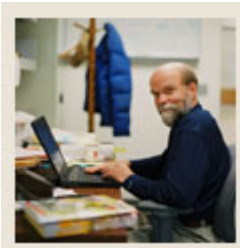
Lesson: Nested IF Statements

Jump to TOC

Nesting

You can nest any IF statement within any other IF statement.

```
/* If the combined score is greater than 1200, the student's scores
are evaluated further.  If the student's verbal score is greater
than 600, a record is inserted into HIGH_VERBAL.  If the math score
is greater than 600, then a record is inserted in HIGH_MATH. */
DECLARE
   lv_pidm          NUMBER;
   lv_id            VARCHAR2(9) := '882993466';
   lv_test_date     DATE := '08-JUN-05';
   lv_sat_verbal    NUMBER(3);
   lv_sat_math      NUMBER(3);
BEGIN
   SELECT swrtest_pidm, swrtest_sat_verbal, swrtest_sat_math
     INTO lv_pidm, lv_sat_verbal, lv_sat_math
     FROM swrtest, swriden
    WHERE swriden_pidm = swrtest_pidm
      AND swriden_id = lv_id
      AND swrtest_test_date = lv_test_date
     AND swriden_change_ind IS NULL;

   IF lv_sat_verbal + lv_sat_math > 1200 THEN
      IF lv_sat_verbal > 600 THEN
         INSERT INTO high_verbal (high_verbal_pidm,
                                  high_verbal_verbal_score,
                                  high_verbal_test_date)
              VALUES (lv_pidm, lv_sat_verbal,
                      lv_test_date);
      END IF;
      IF lv_sat_math > 600 THEN
         INSERT INTO high_math (high_math_pidm, high_math_math_score,
                                high_math_test_date)
              VALUES (lv_pidm, lv_sat_math,
                      lv_test_date);
      END IF;
   END IF;
END;
/
```



Section E: Conditional, Iterative, Sequential Control

Lesson: Iterative Control

Jump to TOC

Loop types

Four types of loops:

- Simple Loops
- Numeric FOR Loops
- WHILE Loops
- Cursor FOR Loops

Simple Loops

The loop is useful when you want to guarantee that the body (or part of body) executes at least one time. The simple loop will execute until an EXIT statement is executed and the value is TRUE.

```
LOOP
   <sequence of statements>
END LOOP;
```

Exiting a simple loop

Exit any loop immediately with the EXIT statement.

```
EXIT WHEN <condition>;
```

Example

```
DECLARE
   lv_counter NUMBER := 1;
BEGIN
   LOOP
      INSERT INTO temp (col1, col2)
         VALUES (lv_counter, sysdate);
      lv_counter := lv_counter + 1;
      EXIT WHEN lv_counter > 10;
   END LOOP;
END;
/
```

Numeric FOR loops

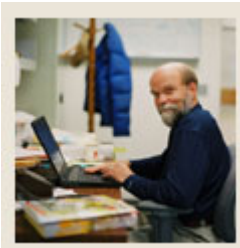
Repeat a sequence of statements a fixed number of times with a Numeric FOR loop.

```
  FOR <index> IN <low_value> .. <high_value>
  LOOP
     <sequence of statements>
  END LOOP;

  BEGIN
     FOR i IN 1..20 LOOP
        INSERT INTO temp (col1)
           VALUES (i);
     END LOOP;
  END;
/
```

Loop index

The index is implicitly of type NUMBER, and cannot be reassigned within the loop. However, it may be used in an expression. It does not need to be declared in the declaration section and only exists for the life of the loop.

```
  DECLARE
     lv_counter NUMBER;
  BEGIN
     FOR loop_index IN 1..20 LOOP
        INSERT INTO temp (col1)
           VALUES (loop_index);
        loop_index := loop_index + 1;  --illegal
        lv_counter := loop_index;      --legal
     END LOOP;
  END;
/
```



WHILE loops

Use the WHILE loop when you want to repeat a sequence of statements until a specific condition is no longer TRUE.

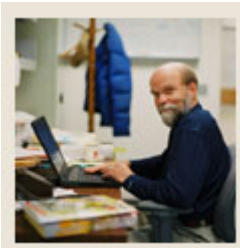
```
WHILE <condition is TRUE>
LOOP
  <sequence of statements>
END LOOP;
```

Example 1

```
DECLARE
   lv_counter  number := 0;
BEGIN
   WHILE lv_counter < 50
   LOOP
      lv_counter := lv_counter + 1;
      INSERT INTO TEMP(col1, col2)
             VALUES (lv_counter, sysdate);
   END LOOP;
END;
/
```

Example 2

```
DECLARE
   lv_end_of_program  BOOLEAN := FALSE;
   lv_counter         number := 0;
BEGIN
   WHILE NOT lv_end_of_program LOOP
      lv_counter := lv_counter + 1;
      INSERT INTO TEMP(col1)
             VALUES (lv_counter);
      IF lv_counter >= 20 THEN
         lv_end_of_program := TRUE;
      END IF;
   END LOOP;
END;
/
```



The GOTO Statement

The GOTO statement performs unconditional branching to a named label.

The syntax for a GOTO statement is:

```
GOTO label_name;
```

The GOTO label is defined as follows:

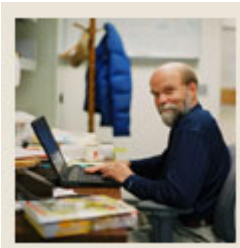
```
<<label_name>>
```

Example

```
IF lv_rating > 80 THEN
   GOTO calc_raise;
END IF;

<<calc_raise>>
IF lv_job_title = 'SALESMAN' THEN
    lv_amount := lv_commission * 0.25;
ELSE
    lv_amount := lv_salary * 0.10;
END IF;
```

Notes

- At least one executable statement must follow a label
- The target label must be in the same scope as the GOTO statement
- The target label must be in the same part of the PL/SQL block as the GOTO statement (GOTO in body cannot go to a label in the exception handler)



Block and loop labels

In addition to using labels for GOTO statements, you can also use labels for blocks and loops.

```
  <<label name>>
  DECLARE
  --declarations
  BEGIN
    --executable statements
  EXCEPTION
    --exception handler
  END label_name;  --must include label name

  <<outer_block>>
  DECLARE
    lv_pidm NUMBER := 1234;
  BEGIN
    /* sub-block */
    DECLARE
       lv_pidm NUMBER := 4321;
    BEGIN
       UPDATE swriden
          SET swriden_last_name = 'SMITH'
        WHERE lv_pidm in (lv_pidm, outer_block.lv_pidm);
    END;
    /* end of sub-block */
  END outer_block;
/
```

EXIT labels

Label an EXIT as a way to specify exits from outer loops.

```
  BEGIN
     <<outer_loop>>
     WHILE NOT lv_end_of_program LOOP
         <<inner_loop>>
         FOR i IN 1..10 LOOP
            EXIT outer_loop WHEN lv_end_of_program = TRUE;
            x := x + 1;
            IF x > 9
            THEN
              lv_end_of_program := TRUE;
            END IF;
        END LOOP inner_loop;
     END LOOP outer_loop;
  END;
/
```

Qualifying variable names with labels

Label names can be used to distinguish between variables with the same name.

Section E: Conditional, Iterative, Sequential Control

Lesson: View Information on Screen

Jump to TOC

DBMS_ OUTPUT

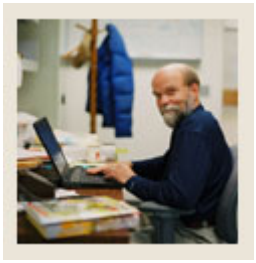
Use the built-in package DBMS_OUTPUT to print to the screen. Before messages can be printed to the screen, the command SET SERVEROUTPUT ON must be entered at the SQL*Plus prompt or in the login.sql file (automatically executed when you log in).

A single string must be passed to the package. It may be a concatenation of strings and variables but must result in a single string.

The package will be discussed in greater detail in Section G.

Example

```
  DECLARE
     lv_numb     number := 0;
  BEGIN
     DBMS_OUTPUT.ENABLE;
     DBMS_OUTPUT.PUT_LINE('Starting Procedure');
     LOOP
          lv_numb := lv_numb + 10;
          EXIT WHEN lv_numb > 100;
     END LOOP;

     DBMS_OUTPUT.PUT_LINE('The value is: '|| lv_numb);
  END;
/
```

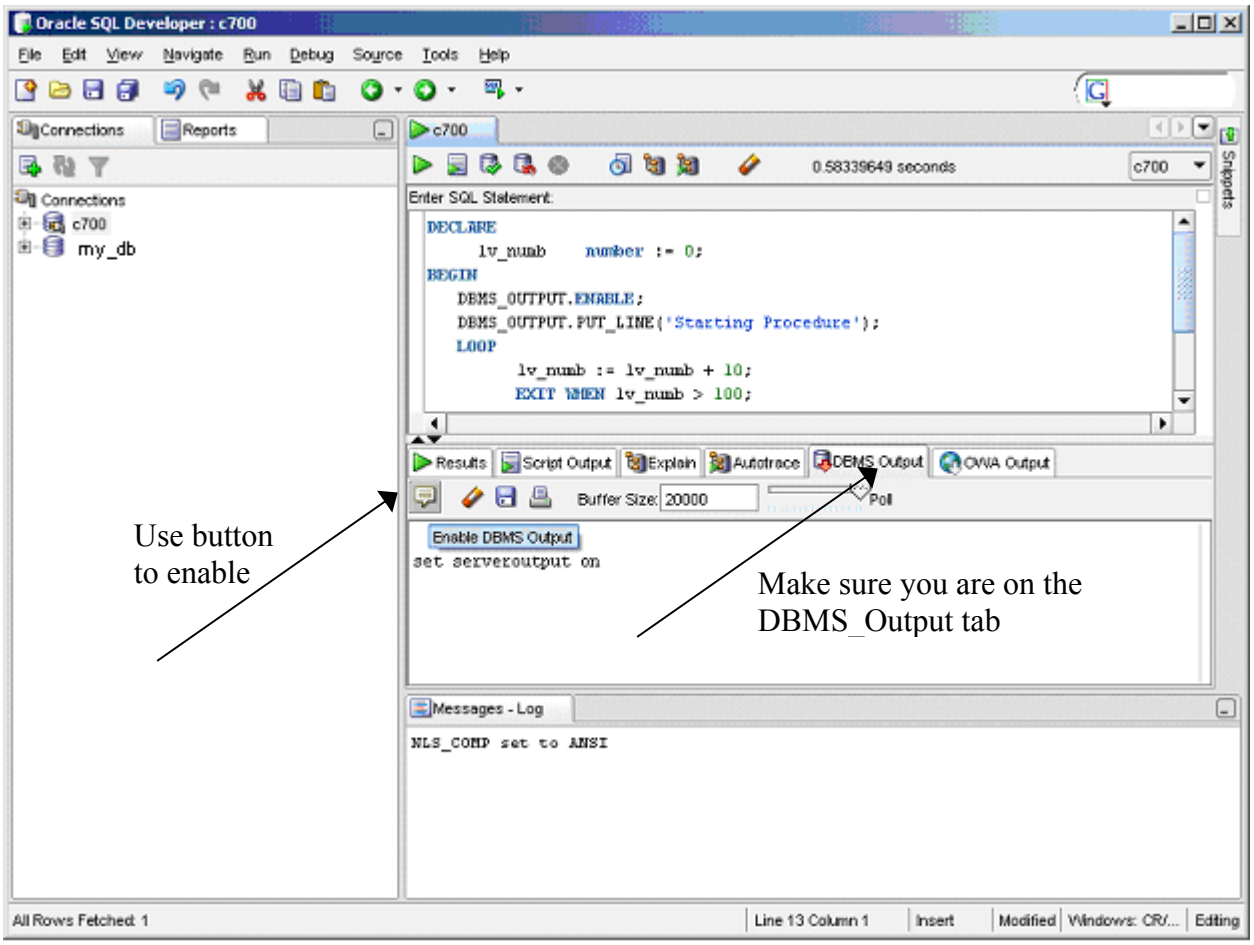


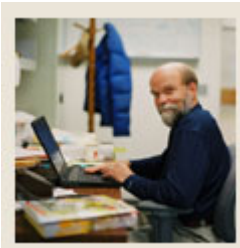
Section E: Conditional, Iterative, Sequential Control

Lesson: View Information on Screen (continued)

Jump to TOC

Enable DBMS_OUTPUT in SQL Developer





Section E: Conditional, Iterative, Sequential Control

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

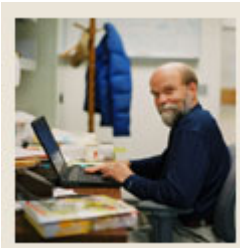
Exercise 1

What are the four types of loops?

Exercise 2

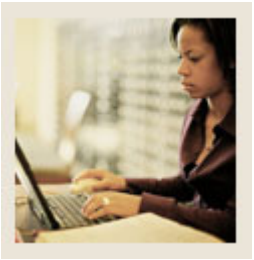
Write a PL/SQL block that will conditionally execute for the following conditions. Your script should ask the user for a value. Use DBMS_OUTPUT to show the value entered and the new value.

- If x = 20, then add 10
- If x = 30, then add 20
- If x = 40, then add 30
- If x = 50, then add 40



Exercise 3

Using a loop, write a PL/SQL block that inserts values and the date that value was calculated into the TEMP table. Values should be between 1 and 20. Check the TEMP table to make sure the values were added.



Section F: Handle PL/SQL Errors

Lesson: Overview

Jump to TOC

Introduction

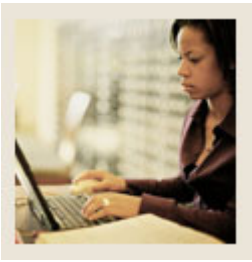
This section provides a basic overview of PL/SQL exception handling.

Objectives

At the end of this section, the attendees will be able to

- understand advantages of exception handling
- identify types of exception handling
- identify PL/SQL error information functions.

Section contents



Section F: Handle PL/SQL Errors

Lesson: Exception Handling

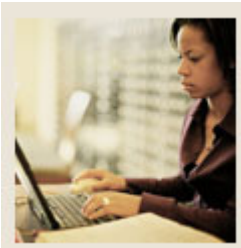
Jump to TOC

Advantages of Exception Handling

- Event-driven handling of errors
- Separation of error-processing code
- Improved reliability of error handling

Types of Exceptions

- Named system exceptions
- Named programmer-defined exceptions
- Unnamed system exceptions



Section F: Handle PL/SQL Errors

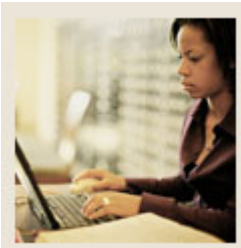
Lesson: Named System Exceptions

Jump to TOC

Common errors

An ORACLE error "raises" an exception automatically. Below are some common errors that have been defined for you.

EXCEPTION NAME	ORACLE ERROR	SQLCODE VALUE
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
STORAGE_ERROR	ORA-06500	-6500
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476



Section F: Handle PL/SQL Errors

Lesson: Named System Exceptions (Continued)

Jump to TOC

Syntax

```
WHEN <exception name> [ OR <exception name> ....] THEN
   <sequence of statements>
...
[WHEN OTHERS THEN  --if used, must be the last handler
   <sequence of statements>]
```

WHEN OTHERS addresses all exceptions not defined within the exception handler.

Example

```
/* Retrieves the last name from swriden based on the ID that the user
enters.  If one row is successfully retrieved, then the ID and last name
are inserted into the TEMP table.  If no rows or too many rows are found,
then the transaction is rolled back and a row containing an error message
is inserted into the TEMP table. */
DECLARE
   lv_student_lname swriden.swriden_last_name%TYPE;
   lv_id_in         swriden.swriden_id%TYPE;
BEGIN
   lv_id_in := &id;
   SELECT swriden_last_name
     INTO lv_student_lname
     FROM swriden
    WHERE swriden_id = lv_id_in
      AND swriden_change_ind IS NULL;
   INSERT INTO temp (col2, col3, message)
      VALUES (lv_id_in, sysdate, lv_student_lname);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      ROLLBACK;
      INSERT INTO temp (col2, col3, message)
         VALUES (lv_id_in, sysdate, ' No Data Found.');
      COMMIT;
   WHEN TOO_MANY_ROWS THEN
      ROLLBACK;
      INSERT INTO temp (col2, col3, message)
         VALUES (lv_id_in, sysdate, 'More than one row found for ID '
                 || lv_id_in);
      COMMIT;
   WHEN OTHERS THEN
      ROLLBACK;
END;
/
```

Section F: Handle PL/SQL Errors

Lesson: Named Programmer-Defined Exceptions

Jump to TOC

Syntax

```
DECLARE
   lv_my_exception  EXCEPTION;
   ...
```

RAISE your exception

```
RAISE lv_my_exception;
```

Notes

- Once an exception is raised manually, it is treated the same way as if it were a predefined internal exception
- Declared exceptions are scoped just like variables
- A user-defined exception is checked for manually and then raised, if appropriate
- Raise can be used with user-defined exceptions as well as built in exceptions

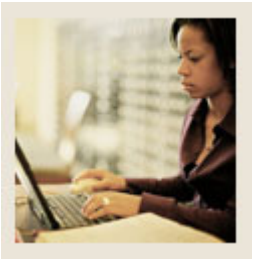
Section F: Handle PL/SQL Errors

Lesson: Named Programmer-Defined Exceptions (Continued)

Jump to TOC

Example

```
DECLARE
   CURSOR C1 IS
      SELECT swrtest_pidm, swrtest_test_date, swrtest_activity_date
        FROM swrtest;
   lv_pidm               NUMBER;
   lv_test_date          DATE;
   lv_activity_date      DATE;
   invalid_activity_date EXCEPTION; /* user name exception */
BEGIN
   OPEN c1;
   /* Exception is within a loop, so that if an test date is later
      than activity date, then the transaction is rolled back and the
      next row is fetched.  For any other error, the loop is exited
      because the error is handled outside the loop
   */
   LOOP
     BEGIN
       EXIT WHEN c1%NOTFOUND;
       FETCH C1 INTO lv_pidm, lv_test_date,
                     lv_activity_date;
         IF lv_test_date > lv_activity_date THEN
           RAISE invalid_activity_date;
         END IF;
         UPDATE swrtest
            SET swrtest_status = 'VALID'
          WHERE swrtest_pidm = lv_pidm;
         COMMIT;
      EXCEPTION
         WHEN invalid_activity_date THEN
            ROLLBACK;
            INSERT INTO temp (col1, col2, message)
              VALUES (lv_pidm, sysdate, 'Activity Date is Later than Test Date.');
            UPDATE SWRTEST
               SET swrtest_status = 'INVALID'
             WHERE swrtest_pidm = lv_pidm;
            COMMIT;
      END;
   END LOOP;
EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK;
      INSERT INTO temp (col1, col2, message)
         VALUES (1234, sysdate, 'ERROR: Routine Failed.');
      COMMIT;
END;
/
```



Section F: Handle PL/SQL Errors

Lesson: Exception Propagation

Jump to TOC

Steps

The following are the steps involved in exception propagation.

Step	Action
1	The current block is searched for a handler. If not found, go to step 2.
2	If an enclosing block is found, it is searched for a handler.
3	Steps 1 and 2 are repeated until either there are no more enclosing blocks, or a handler is found. • If there are no more enclosing blocks, the exception is passed back to the calling environment • If a handler is found, it is executed. When done, the block in which the handler was found is terminated, and control is passed to the enclosing block (if one exists), or to the environment (if there is no enclosing block)

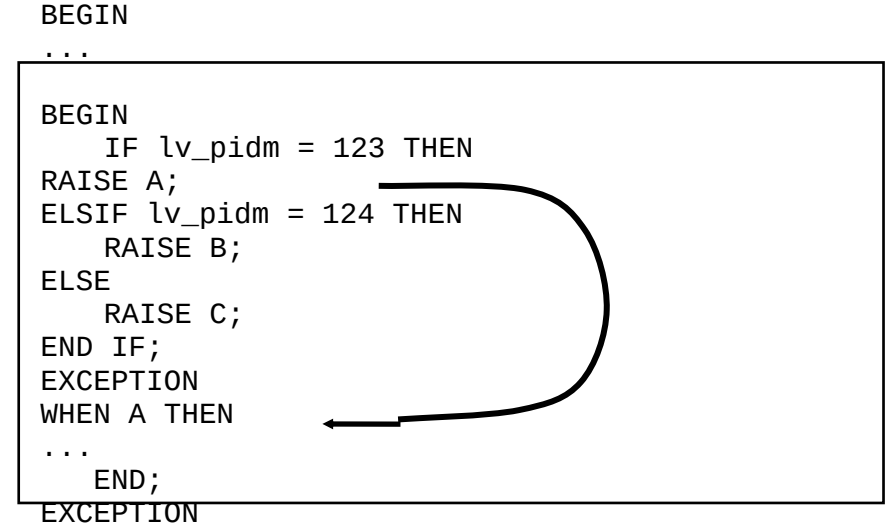
Notes

- Only one handler per block may be active at a time
- If an exception is raised in a handler, the search for a handler for the new exception begins in the enclosing block of the current block

Example 1

Example - When A is Raised

Diagram



```
BEGIN
...

BEGIN
    IF lv_pidm = 123 THEN
RAISE A;
ELSIF lv_pidm = 124 THEN
    RAISE B;
ELSE
    RAISE C;
END IF;
EXCEPTION
WHEN A THEN
...
   END;
EXCEPTION
          WHEN B THEN
          ...
END;
/
```

Example 2

Example - When B is Raised

Diagram

```
  BEGIN
  ...
  BEGIN
  IF lv_pidm = 123 THEN
     RAISE A;
  ELSIF lv_pidm = 124 THEN
     RAISE B;
  ELSE
     RAISE C;
  END IF;
  EXCEPTION
  WHEN A THEN
                ...
  END;
  EXCEPTION
     WHEN B THEN
        ...
  END;
/
```

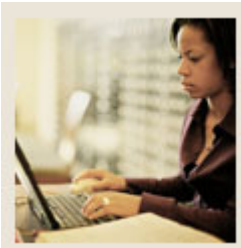
Example 3

Example - When C is Raised

Diagram

```
BEGIN
...
BEGIN
IF lv_pidm = 123 THEN
   RAISE A;
ELSIF lv_pidm = 124 THEN
   RAISE B;
ELSE
   RAISE C;
END IF;
EXCEPTION
WHEN A THEN
              ...
END;
EXCEPTION
WHEN B THEN
           ...
END;
```

Exception C has no handler and will result in a runtime unhandled exception.



Section F: Handle PL/SQL Errors

Lesson: Unnamed System Exceptions

Jump to TOC

PRAGMA

What do you do when you want to trap an error that has not been predefined? You will need to define the error yourself. Exceptions may only be handled by name (not ORACLE number). To handle undefined errors, use PRAGMA.

Syntax

```
PRAGMA EXCEPTION_INIT (<user_defined_exception_name>,
<ORACLE_error_number>);
```

Example

```
DECLARE
   invalid_session EXCEPTION;
   PRAGMA          EXCEPTION_INIT (invalid_session, -22);

BEGIN
   INSERT INTO swriden (swriden_pidm, swriden_id,
               swriden_last_name, swriden_activity_date)
      VALUES (1234, '56', 'Peterson',SYSDATE);
EXCEPTION
  WHEN invalid_session THEN
     INSERT INTO TEMP(COL1, COL2, MESSAGE)
        VALUES (-22, SYSDATE, 'Invalid session ID.  Access Denied');
  WHEN OTHERS THEN
     ROLLBACK;
END;
/
```

Section F: Handle PL/SQL Errors

Lesson: SQLCODE and SQLERRM

Jump to TOC

Viewing error information

The WHEN OTHERS exception handles all unspecified errors. However, even if you want to generically handle all unspecified errors (such as rolling back changes made to the database), you will want to know what error occurred. SQLCODE and SQLERRM will provide error information to you.

SQLCODE

Returns the ORACLE error number of the exceptions, or 1 if it was a user-defined exception.

SQLERRM

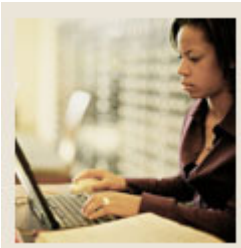
Returns the ORACLE error message associated with the current value of SQLCODE.
Can also use any ORACLE error number as an argument.

No exception active

If no exception is active:

- SQLCODE = 0
- SQLERRM= 'normal, successful completion'

SQLCODE and SQLERRM cannot be used within a SQL statement.



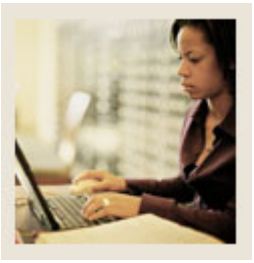
Section F: Handle PL/SQL Errors

Lesson: SQLCODE and SQLERRM (Continued)

Jump to TOC

Example

```
DECLARE
   invalid_session EXCEPTION;
   PRAGMA          EXCEPTION_INIT (invalid_session, -22);
   lv_sqlcode      NUMBER;
   lv_sqlerrm      VARCHAR2(55);
BEGIN
   INSERT INTO swriden (swriden_pidm, swriden_id,
                        swriden_last_name,swriden_activity_date)
      VALUES (1234, '56', 'Peterson',SYSDATE);
EXCEPTION
  WHEN invalid_session THEN
     INSERT INTO TEMP(COL1, COL2, MESSAGE)
        VALUES (-22, SYSDATE,
                'Invalid session ID.  Access Denied');
  WHEN OTHERS THEN
      lv_sqlcode := SQLCODE;
      lv_sqlerrm := SUBSTR(SQLERRM, 1, 55);
      ROLLBACK;
      INSERT INTO temp (col1, col2, message)
         VALUES (lv_sqlcode, sysdate, lv_sqlerrm);
      COMMIT;
END;
/
```



Did the process really run?

If you are not checking for errors or do not have SERVEROUTPUT turned on to show you the error you may be fooled into thinking your process finished properly when it did not.

Many PL/SQL programs will end with the phrase 'PL/SQL program completed successfully' when there was actually a problem encountered in the code.

```
SQL>  DECLARE
  2     lv_over50  number := &my_number;
  3  BEGIN
  4     IF lv_over50 < 50 THEN
  5        DBMS_OUTPUT.PUT_LINE('Your number must be greater than 50;');
  6     ELSE
  7        DBMS_OUTPUT.PUT_LINE('Thank you for a number 50 or larger');
  8     END IF;
  9* END;
SQL> /
Enter value for my_number: 22

PL/SQL procedure successfully completed.

SQL> set serveroutput on
SQL> /
Enter value for my_number: 22

Your number must be greater than 50;

PL/SQL procedure successfully completed.
```

Section F: Handle PL/SQL Errors

Lesson: Forcing Program Abort

Jump to TOC

RAISE_APPLICATION_ERROR

You can force your program to abort and provide a user-created error by using a built-in called RAISE_APPLICATION_ERROR. The procedure takes two required arguments:

- An error number in the range -20000 to -20999
- An error message in the form of a string or variable

```
SQL>  DECLARE
  2     lv_over50  number := &my_number;
  3  BEGIN
  4     IF lv_over50 < 50 THEN
  5        raise_application_error('-20999','Your number must be greater
than 50');
  6     ELSE
  7        DBMS_OUTPUT.PUT_LINE('Thank you for a number 50 or larger');
  8     END IF;
  9* END;
SQL> /
Enter value for my_number: 22
DECLARE
*
ERROR at line 1:
ORA-20999: Your number must be greater than 50
ORA-06512: at line 5
```

Section F: Handle PL/SQL Errors

Lesson: Debugging

Jump to TOC

Developing a debugging plan

When you write simple PL/SQL programs, the error messages might be enough to let you know how to fix bugs. However, when your programs become more complicated and you have stored subprograms being called, it helps to have a game plan as to how to narrow your search.

Define What Went Wrong

When a PL/SQL program does not work, you will need to determine the specifics. Did Oracle actually return an error? Did the program complete successfully, but return/calculate the wrong value?

Reduce the Amount of Code

If you are dealing with a large program, it is helpful to comment out code to narrow your search. Try commenting out a large section and running the program again. If the error still occurs, then you can rule out the entire section that is commented out as being the problem.

Establish a Test Environment

Make sure that you are testing in an environment that allows you to experiment. Sometimes it is helpful to create 'table clones'.

For instance, you are inserting rows into the SWRIDEN table, and the values don't seem to be inserting correctly. However, the rows you are inserting are being 'mixed in' with the rows that were initially there. To avoid confusion, create a clone of the table, but omit the rows. This way, you know that the rows that are inserted are only the rows inserted from your program.

```
CREATE TABLE swriden_temp
   AS SELECT *
        FROM swriden
       WHERE 1=2;
```

Evaluate Variable Values

- Print to the screen using DBMS_OUTPUT
- Insert into a test table
- Use variables that change based on the section of code you are in so you know which section is causing the problem

Section F: Handle PL/SQL Errors

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

What are three advantages of using PL/SQL Error Handling?

Exercise 2

What are three types of PL/SQL exceptions?

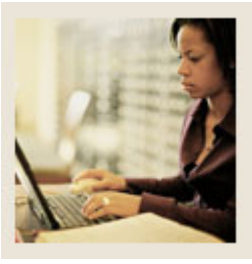
Exercise 3

Identify the type of PL/SQL exceptions in the following examples:

```
… DECLARE my_exception  Exception;
```

```
…PRAGMA EXCEPTION_INIT my_exception, 1025;
```

```
DECLARE…
BEGIN…
…….
EXCEPTION
WHEN OTHERS THEN
    ROLLBACK;
END;
end
```



Exercise 4

Redo Exercise 2 from Section E to raise a user-defined exception if the number entered is not 20, 30, 40, or 50. Process the error in an exception handler and display an appropriate message.

Section G: Cursors, Records, and Tables

Lesson: Overview

Jump to TOC

Introduction

A PL/SQL cursor allows you to fetch and process your data, one row at a time. You can also use a PL/SQL table, which is similar to an array. A PL/SQL table will allow you to easily search forward and backward.

Objectives

Upon completion of this section, each attendee will be able to

- identify the two types of cursors supported by PL/SQL
- outline the procedural steps for using explicit cursors in PL/SQL blocks
- demonstrate comprehension of PL/SQL cursors by writing a simple PL/SQL procedure using explicit cursors
- use implicit cursor attributes
- create and reference PL/SQL tables.

Section contents

Associated cursor

SQL Statements with Associated Cursor

- INSERT
- UPDATE
- DELETE
- SELECT...INTO
- COMMIT
- ROLLBACK

Cursor types

Two types of cursors:

- EXPLICIT
  - Multiple row SELECT Statements
- IMPLICIT
  - All INSERT Statements
  - All UPDATE Statements
  - All DELETE Statements
  - Single row SELECT…INTO Statements

Handling multiple returned rows

The set of rows returned by a query can consist of zero, one, or many rows, depending upon the number of rows that meet the query's search condition.

When a query returns multiple rows, a cursor can be explicitly defined to:

- Process beyond the first row returned by the query
- Keep track of which rows are currently being processed

Cursor operations

- Declare the cursor
- Open the cursor
- Fetch data from the cursor
- Close the cursor

Syntax

```
DECLARE
   CURSOR <cursor name>
      IS <regular_select_statement>;
```

Scoping declared cursors

Declared cursors are scoped just as variables are.

```
DECLARE
   lv_pidm NUMBER(8);

   CURSOR c1 IS
         SELECT swriden_pidm
           FROM swriden
          WHERE swriden_change_ind IS NULL;
BEGIN...
```

Section G: Cursors, Records, and Tables

Lesson: Open Cursors

Jump to TOC

Open cursors

Open the cursor to process the SELECT statement and store the returned rows in the cursor.

```
OPEN <cursor name>;
```

Example

```
OPEN c1;
```

Functions

- Evaluates the SELECT statement associated with the cursor
- Allocates the resources used by ORACLE to process the query
- Identifies the active set

Fetch data

Fetch data from the cursor and store it in specified variables.

```
FETCH <cursor name> INTO <var1, var2...>;
```

Example

```
FETCH c1 INTO lv_pidm;
```

Restrictions

There must be exactly one INTO variable for each column selected by the SELECT statement. The first column gets assigned to var1, the second assigned to var2, etc. These variables must be declared in the declaration section.

%ROWTYPE

If fetching all columns of a table the %ROWTYPE operator can be used with a single variable. When using this type of declaration the individual columns can be referenced as <cursor_name>.<column_name>.

```
DECLARE
      lv_swriden_row    swriden%ROWTYPE;
      CURSOR c1 IS
             SELECT *
               FROM swriden
              WHERE swriden_change_ind is null;
BEGIN
   OPEN c1;
   LOOP
     FETCH c1 INTO lv_swriden_row;
     EXIT WHEN c1%NOTFOUND;
        INSERT INTO SWRIDEN_HISTORY
        VALUES (lv_swriden_row.swriden_pidm,
        lv_swriden_row.swriden_id,
        lv_swriden_row.swriden_last_name,
        lv_swriden_row.swriden_first_name,
        lv_swriden_row.swriden_mi,
        lv_swriden_row.swriden_change_ind,
        lv_swriden_row.swriden_activity_date);
   END LOOP;
END;
/
```

Section G: Cursors, Records, and Tables

Lesson: Close Cursors

Jump to TOC

Close cursors

Close the cursor to free up resources.

```
CLOSE <cursor name>;
```

Example

```
CLOSE c1;
```

Functions

- Marks resources held by opened cursor as reusable by other processes
- No more rows can be fetched from a closed cursor

%FOUND Attribute

After a cursor or cursor variable is opened but before the first fetch, %FOUND yields NULL. Thereafter, it yields TRUE if the last fetch returned a row, or FALSE if the last fetch failed to return a row.

```
FETCH swriden_cursor INTO lv_last_name, lv_first_name;
WHILE swriden_cursor%FOUND LOOP
   FETCH swriden_cursor INTO lv_last_name, lv_first_name;
    /* --data processing here */
END LOOP;
```

%NOTFOUND Attribute

Logical opposite of %FOUND.

```
LOOP
  FETCH swriden_cursor INTO lv_last_name, lv_first_name;
  EXIT WHEN swriden_cursor%NOTFOUND;
   /* --data processing here */
END LOOP;
```

%ROWCOUNT Attribute

When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far.

```
LOOP
  FETCH swriden_cursor INTO lv_last_name, lv_first_name;
  EXIT WHEN swriden_cursor%NOTFOUND OR
            swriden_cursor%ROWCOUNT > 100;
   /* --data processing here */
END LOOP;
```

%ISOPEN Attribute

Yields TRUE if its cursor or cursor variable is open; otherwise, yields FALSE.
A cursor must be closed before it can be reopened, so you can use this attribute to test whether the cursor is open or not.

```
IF swriden_cursor%ISOPEN THEN
   FETCH swriden_cursor INTO lv_last_name, lv_first_name;
ELSE
   OPEN swriden_cursor;
   FETCH swriden_cursor INTO lv_last_name, lv_first_name;
END IF;
```

Example

```
/* For each student, the pidm and average gpa is selected from the
SWRREGS table. The rows are ordered so that the student with the
highest gpa is in the first row selected.  Each row is fetched and
then assigned a rank.  Then the PIDM and rank is inserted into the
SWRRANK table. */

DECLARE
   lv_rank NUMBER := 0;
   lv_pidm NUMBER := 0;
   lv_gpa  NUMBER(5,2) := 0;
   CURSOR c1 is
       SELECT swrregs_pidm, AVG(swrregs_gpa)
         FROM swrregs
        GROUP BY swrregs_pidm
        ORDER BY AVG(swrregs_gpa) DESC;
BEGIN
  OPEN c1;
  LOOP
     FETCH c1 INTO lv_pidm, lv_gpa;
     EXIT WHEN c1%NOTFOUND;
     lv_rank := lv_rank + 1;
     INSERT INTO swrrank (swrrank_pidm, swrrank_class_rank,
                          swrrank_activity_date)
            VALUES (lv_pidm, lv_rank, sysdate);
  END LOOP;
  IF c1%ISOPEN then
     CLOSE c1;
  END IF;
END;
```

WHERE CURRENT OF statement

Reference the current row with the WHERE CURRENT OF statement.

```
WHERE CURRENT OF <cursor_name>
```

The cursor must be declared with a FOR UPDATE clause when you will be updating, or deleting from the database table.

Example

```
   DECLARE
      swriden_row swriden%ROWTYPE;
      CURSOR c1 is
         SELECT *
           FROM swriden
          WHERE swriden_change_ind IS NOT NULL
            FOR UPDATE;
   BEGIN
      OPEN c1;
      LOOP
         FETCH c1 INTO swriden_row;
         EXIT WHEN c1%NOTFOUND;
         INSERT INTO swriden_temp
            (swriden_pidm, swriden_id, swriden_last_name,
                swriden_first_name, swriden_change_ind,
                swriden_activity_date)
          VALUES(swriden_row.swriden_pidm, swriden_row.swriden_id,
                swriden_row.swriden_last_name,
                swriden_row.swriden_first_name,
                swriden_row.swriden_change_ind, SYSDATE);
         DELETE FROM swriden
            WHERE CURRENT OF c1;
      END LOOP;
      CLOSE c1;
   END;
/
```

Section G: Cursors, Records, and Tables

Lesson: Cursor FOR Loops

Jump to TOC

Cursor FOR loops

Specify a sequence of statements to be repeated once for each row that is returned by the cursor with the Cursor FOR Loop.

```
FOR <record_name> IN <cursor_name> LOOP
   --statements to be repeated go here
END LOOP;
```

Restrictions

- Cursor FOR loops are similar to Numeric FOR loops
- Cursor FOR loops specify a set of rows from a table using the cursor's name. Numeric FOR loops specify an integer range
- A Cursor FOR loop record takes on the values of each row
- A Numeric FOR loop index takes on each value in the range
- Record_name is implicitly declared as:
  ```
  record_name cursor%ROWTYPE;
  ```
- To reference an element of the record, use the record_name.column_name notation
- Functions in the select statement must have column alias'

Section G: Cursors, Records, and Tables

Lesson: Conceptual Cursor Loop Model

Jump to TOC

Model

- When a cursor is initiated, an implicit OPEN cursor_name is executed
- For each row that satisfies the query associated with the cursor, an implicit FETCH is executed into the components of record_name
- When there are no more rows left to FETCH, an implicit CLOSE cursor_name is executed and the loop is exited

Example

```
  DECLARE
     lv_rank NUMBER := 0;
     CURSOR c1 is
        SELECT swrregs_pidm, AVG(swrregs_gpa) avg_gpa
          FROM swrregs
      GROUP BY swrregs_pidm
      ORDER BY AVG(swrregs_gpa) DESC;
  BEGIN
     /* --an implicit open is done here */
     FOR cursor_row IN c1 LOOP
      /* --an implicit fetch is done here */
        lv_rank := lv_rank + 1;
        IF cursor_row.avg_gpa > 3.0 THEN
           /* could not use 'cursor_row.avg(swrregs_gpa)' here
              must have a column alias */
           INSERT INTO swrrank (swrrank_pidm, swrrank_class_rank,
                                swrrank_activity_date)
               VALUES (cursor_row.swrregs_pidm, lv_rank, sysdate);
        END IF;
     END LOOP;  /* --an implicit close is done here */
  END;
/
```

Associated statements

- All INSERT Statements
- All UPDATE Statements
- All DELETE Statements
- All SELECT..INTO Statements

Restrictions

An implicit cursor is called the "SQL" cursor; it stores information concerning the processing of the last SQL statement not associated with an explicit cursor.

OPEN, FETCH, and CLOSE do not apply.

All cursor attributes apply.

SQL% NOTFOUND

SQL%NOTFOUND evaluates to TRUE if the most recently executed SQL statement affects no rows.

```
DECLARE
BEGIN
   UPDATE swbpers
      SET swbpers_ssn = '123456789'
    WHERE swbpers_pidm = 12340;
   IF SQL%NOTFOUND THEN
      INSERT INTO swbpers (swbpers_pidm, swbpers_ssn,
                           swbpers_activity_date)
          VALUES (12340, '123456789', SYSDATE);
   END IF;
END;
```

SQL%FOUND

SQL%FOUND evaluates to TRUE if the most recently executed SQL statement affects one or more rows.

SQL% ROWCOUNT

SQL%ROWCOUNT evaluates to the number of rows affected by a DELETE, UPDATE, or INSERT.

```
DECLARE
   lv_count      NUMBER;
BEGIN
   INSERT INTO swriden_temp
      SELECT * FROM swriden
         WHERE swriden_change_ind IS NULL;
   lv_count := SQL%ROWCOUNT;
   INSERT INTO temp (col1, col3, message)
      VALUES (lv_count, SYSDATE,'Rows copied to SWRIDEN_TEMP');
END;
/
```

Section G: Cursors, Records, and Tables

Lesson: PL/SQL Records

Jump to TOC

%ROWTYPE

You can use %ROWTYPE to represent multiple variables that relate back to all the columns in a particular table (as discussed above), which is particularly helpful when selecting all columns from a table into host variables.

```
  SET SERVEROUTPUT ON
  DECLARE
     lv_swriden_rec swriden%ROWTYPE;
     CURSOR swriden_cursor IS
        SELECT *
          FROM swriden;
  BEGIN
     DBMS_OUTPUT.ENABLE;
     OPEN swriden_cursor;
     LOOP
        FETCH swriden_cursor INTO lv_swriden_rec;
        EXIT WHEN swriden_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(lv_swriden_rec.swriden_first_name
                             ||' '||lv_swriden_rec.swriden_last_name);
     END LOOP;
     CLOSE swriden_cursor;
  END;
/
```

Composite types

With %ROWTYPE, you cannot alter the variables or datatypes. If you need more flexibility, you can define your own composite types, and then create records that are of that record type.

```
  DECLARE
     TYPE personrectyp IS RECORD
        (pidm      NUMBER(8),
         id        VARCHAR2(9),
         full_name VARCHAR2(30));

     person_record personrectyp;
  …
```

Syntax

```
TYPE record_type IS RECORD (
  Field1 type1 [NOT NULL] [:= expr1],
  Field2 type2 [NOT NULL] [:= expr2],
  …
  Fieldn typen [NOT NULL] [:= exprn]);
```

Notes

- A record can have as many fields as desired
- Multiple records can use the same type

```
DECLARE
   TYPE personrectyp IS RECORD
      (pidm      NUMBER(8),
       id        VARCHAR2(9),
       full_name VARCHAR2(60));

   student_record  personrectyp;
   employee_record personrectyp;
…
```

Declared record

Once the record is declared, you can precede the variable name with the record name.

```
BEGIN
   Student_record.id := '123456789';
   Student_record.pidm := 1;
   Student_record.full_name := 'Sam Smith';
…
END;
/
```

Record assignment

In order for one record to be assigned to another, both records must be of the same type.

```
IF student_is_employed THEN
    employee_record := student_record;
END IF;
```

Assignment via SELECT INTO

A record can also be assigned with a SELECT statement.

```
SET SERVEROUTPUT ON
DECLARE
   TYPE personrectyp IS RECORD
      (pidm      NUMBER(8),
       id        VARCHAR2(9),
       full_name VARCHAR2(60));
   student_record  personrectyp;
   employee_record personrectyp;
BEGIN
   DBMS_OUTPUT.ENABLE;
   SELECT swriden_pidm, swriden_id, swriden_first_name||
          ' '||swriden_mi||' '||swriden_last_name
     INTO student_record
     FROM swriden
    WHERE swriden_change_ind IS NULL
      AND swriden_pidm = 12342;
   DBMS_OUTPUT.PUT_LINE ('Pidm: ' ||student_record.pidm ||
                         '  ID: ' ||student_record.id ||
                         ' Name: ' ||student_record.full_name);
END;
/
```

Section G: Cursors, Records, and Tables

Lesson: Tables

Jump to TOC

PL/SQL tables

Data can be selected and manipulated one row at a time through PL/SQL cursors. However, cursors are intended to be used in a forward manner, meaning that the first row is fetched and processed, and then each subsequent row is fetched and processed. After all rows are fetched, the cursor is closed. Cursors are somewhat inflexible in the sense that processing cannot easily jump forward and backward. PL/SQL tables are one possible solution to this problem.

There are two types of PL/SQL tables, nested tables and associative arrays (formerly known as index-by tables). The associative arrays type will be used in this course.

Declaring PL/SQL tables

PL/SQL tables are similar to arrays in C. However, in order to declare a PL/SQL table, you need to define the table type, and then declare a variable of this type.

```
DECLARE
   TYPE pidm_table IS TABLE OF number(8)
      INDEX BY BINARY_INTEGER;
   lv_pidm_tbl pidm_table;
```

Syntax

```
TYPE tabletype is TABLE OF type INDEX BY BINARY_INTEGER;
```

Refer to table elements

Once the type and the variable are declared, you can refer to an individual element in the PL/SQL table by using the syntax:

```
Tablename(index)
```

Table indexes

The index is a variable of either type BINARY_INTEGER, or a variable or expression that can be converted to a BINARY_INTEGER.

```
BEGIN
   lv_pidm_tbl(1) := 123;
   lv_pidm_tbl(2) := 124;
…
END;
/
```

Section G: Cursors, Records, and Tables

Lesson: Tables vs. Arrays

Jump to TOC

PL/SQL Tables

Typical arrays rely on the fact that the index has a particular sequence to it (1,2,3,4). However, a PL/SQL table is more similar to an Oracle table, which has a KEY column and a VALUE column. The KEY is the binary integer used to look up the table value.

What does this mean? As long as the key is a binary integer, anything goes!
Your index does not have to be in sequential order.

```
BEGIN
 …
   lv_pidm_tbl(-10) := 123;
   lv_pidm_tbl(5) := 124;
   IF lv_pidm_tbl(-10) > lv_pidm_tbl(5) THEN
   …
   END IF;
END;
/
```

Composite types

Prior to PL/SQL version 2.3, tables could only hold scalar types (VARCHAR2, NUMBER, etc.). This meant that a separate table definition was required for each database field. However, 2.3 and above allows composite types, or tables of records.

```
DECLARE
   Type persontabtype IS TABLE OF swriden%ROWTYPE
      INDEX BY BINARY_INTEGER;
    employee_rec persontabtype;
BEGIN
   SELECT *
     INTO employee_rec(10)
     FROM swriden
    WHERE swriden_pidm = 10;
   ...
END;
```

Referring to fields within records

To refer to field within the record, use the following syntax:

```
Table(index).field
employee_rec(10).swriden_last_name = 'Smith';
employee_rec(10).swriden_first_name = 'Sam';
```

Section G: Cursors, Records, and Tables

Lesson: Table Attributes

Jump to TOC

Table attributes

Tables of records can be manipulated using table attributes.

Attribute	Type Returned	Description
COUNT	NUMBER	Returns the number of rows in the table.
DELETE	N/A	Deletes rows in a table.
EXISTS	BOOLEAN	Returns TRUE if the specified entry exists in the table.
FIRST	BINARY_INTEGER	Return the index of the first row in the table.
LAST	BINARY_INTEGER	Returns the index of the last row in the table.
NEXT	BINARY_INTEGER	Returns the index of the next row in the table after the specified row.
PRIOR	BINARY_INTEGER	Returns the index of the previous row in the table before the specified row.

COUNT

```
DECLARE
   Type persontabtype IS TABLE OF swriden%ROWTYPE
      INDEX BY BINARY_INTEGER;
    employee_rec persontabtype;
    CURSOR swriden_cursor IS
       SELECT *
         FROM swriden
        WHERE swriden_change_ind IS NULL;
    lv_counter       NUMBER(6) := 0;
    lv_total_records NUMBER(6);
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   OPEN swriden_cursor;
   LOOP
      EXIT WHEN swriden_cursor%NOTFOUND;
      lv_counter := lv_counter + 1;
      FETCH swriden_cursor INTO employee_rec(lv_counter);
   END LOOP;
   CLOSE swriden_cursor;
    lv_total_records := employee_rec.count;
         DBMS_OUTPUT.PUT_LINE('Total records: '||
                              lv_total_records);

END;
/
Total records: 27
```

DELETE

Removes rows from a PL/SQL table.

- Table.DELETE will remove all rows from a table.
- Table.DELETE(i) will remove the row with index i.
- Table.DELETE(i,j) will remove all rows with indices between i and j.

```
  DECLARE
     Type persontabtype IS TABLE OF swriden%ROWTYPE
        INDEX BY BINARY_INTEGER;
     employee_rec persontabtype;
  BEGIN
     SELECT *
       INTO employee_rec(10)
       FROM swriden
      WHERE swriden_pidm = 12340
        AND swriden_change_ind IS NULL;
  /* Delete record 10 from table employee_rec. */
     employee_rec.DELETE(10);
END;


  DECLARE
     Type persontabtype IS TABLE OF swriden%ROWTYPE
        INDEX BY BINARY_INTEGER;
      employee_rec persontabtype;
  BEGIN
     /* Select records for table employee_rec */
     FOR I in 1..10 LOOP
     SELECT *
       INTO employee_rec(I)
       FROM swriden
      WHERE swriden_pidm = 12350
        AND swriden_change_ind IS NULL;
     END LOOP;
  /* Delete records with indices between 6 and 10 from table
  employee_rec. */
     employee_rec.DELETE(6,10);

     /* Deletes all records from table employee_rec */
     employee_rec.DELETE;
END;
```

EXISTS

Returns TRUE if a row with index i is in the table, and FALSE if the row does not exist in the table. This attribute is useful when you want to make sure a record exists before referring to a non-existent element.

```
DECLARE
   TYPE persontabtype IS TABLE OF swriden%ROWTYPE
      INDEX BY BINARY_INTEGER;
   employee_rec   persontabtype;
   CURSOR swriden_cursor IS
      SELECT *
        FROM swriden
       WHERE swriden_change_ind IS NULL;

   lv_counter        NUMBER (6)    := 0;
BEGIN
   OPEN swriden_cursor;

   LOOP
      EXIT WHEN swriden_cursor%NOTFOUND;
      lv_counter := lv_counter + 1;
      FETCH swriden_cursor INTO employee_rec (lv_counter);
   END LOOP;

   CLOSE swriden_cursor;

   IF employee_rec.EXISTS(10) THEN
      employee_rec.DELETE(10);
   END IF;
END;
/
```

FIRST and LAST

Return the index of the first and last rows in the PL/SQL table.

```
SET SERVEROUTPUT ON
DECLARE
   Type persontabtype IS TABLE OF swriden%ROWTYPE
      INDEX BY BINARY_INTEGER;
    employee_rec persontabtype;
    CURSOR swriden_cursor IS
       SELECT *
         FROM swriden
        WHERE swriden_change_ind IS NULL;
    lv_counter NUMBER(6) := 0;
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   OPEN swriden_cursor;
   LOOP
      EXIT WHEN swriden_cursor%NOTFOUND;
      lv_counter := lv_counter + 1;
      FETCH swriden_cursor INTO employee_rec(lv_counter);
   END LOOP;
   CLOSE swriden_cursor;
   DBMS_OUTPUT.PUT_LINE('Total records: '|| employee_rec.count);
   employee_rec.DELETE(1);
   employee_rec.DELETE(3);
   employee_rec.DELETE(10);
   DBMS_OUTPUT.PUT_LINE('After deleting rows...');
   DBMS_OUTPUT.PUT_LINE(' First record: '|| employee_rec.first);
   DBMS_OUTPUT.PUT_LINE('  Last record: '|| employee_rec.last);
   DBMS_OUTPUT.PUT_LINE('Total records: '|| employee_rec.count);
END;
/

Total records: 27
After deleting rows...
First record: 2
Last record: 27
Total records: 24
```

NEXT and PRIOR

Return the index of the next and previous element in the PL/SQL table.

```
DECLARE
    Type persontabtype IS TABLE OF swriden%ROWTYPE
      INDEX BY BINARY_INTEGER;
    employee_rec persontabtype;
    CURSOR swriden_cursor IS
       SELECT *
         FROM swriden
        WHERE swriden_change_ind IS NULL;
    lv_counter          NUMBER(6) := 0;
    lv_current_rec      NUMBER(6);
    lv_total_recs       NUMBER(6);
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   OPEN swriden_cursor;
   LOOP
      EXIT WHEN swriden_cursor%NOTFOUND;
      lv_counter := lv_counter + 1;
      FETCH swriden_cursor INTO employee_rec(lv_counter);
   END LOOP;
   CLOSE swriden_cursor;
   DBMS_OUTPUT.PUT_LINE('Total records: '||employee_rec.count);
   employee_rec.DELETE(1);
   employee_rec.DELETE(3);
   employee_rec.DELETE(10);
   DBMS_OUTPUT.PUT_LINE('After deleting rows...');
   DBMS_OUTPUT.PUT_LINE(' First record: '|| employee_rec.first);
   DBMS_OUTPUT.PUT_LINE('  Last record: '|| employee_rec.last);
   DBMS_OUTPUT.PUT_LINE('Total records: '|| employee_rec.count);
   lv_current_rec := employee_rec.FIRST;
   lv_total_recs := employee_rec.count;

FOR i IN 1 .. lv_total_recs LOOP
    DBMS_OUTPUT.PUT_LINE(
       employee_rec(lv_current_rec).swriden_first_name ||' '||
       employee_rec(lv_current_rec).swriden_last_name);
    lv_current_rec := employee_rec.NEXT(lv_current_rec);
  END LOOP;
END;
/
```

(results on next page…)

NEXT and PRIOR, continued

```
Total records: 27
After deleting rows...
First record: 2
Last record: 27
Total records: 24
Robert Smith
Sandy Jones-Erickson
Ralph Erickson
Susan Erickson
Nancy White
Joan Marx
Stephanie Clifford
Michelle Dukes
……….
```

Section G: Cursors, Records, and Tables

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

Write a PL/SQL script using a cursor to display the id, last_name and first_name from SWRIDEN.

Exercise 2

Write a PL/SQL script that prompts for a pidm, and selects all columns from the person table, SWBPERS, based upon that pidm. Rather than explicitly declaring all the host variables, use %ROWTYPE. Select the columns by using the SELECT INTO statement. Display the SSN and birth date variables using DBMS_OUTPUT.PUT_LINE. If no record is found, then display an error message to the user.

Exercise 3

Create a PL/SQL script which selects the pidm, id, first_name || last_name (column alias of 'name') and change indicator from SWRIDEN. Select both current rows (change_ind is null) and non-current rows from SWRIDEN (change_ind is NOT null). Sort by pidm and change indicator.

Write each pidm, id, and name using the DBMS_OUTPUT package. If the change indicator is null, specify that the record is 'Current' when writing the line. If the change indicator is not null, specify 'Historical'. Run your script.

Exercise 4

Starting with Exercise 3, using an IF statement, alter your script so that the person information (SSN and birth date) is displayed for current rows and is not displayed for the historical records (change indicator is not null).

Exercise 5

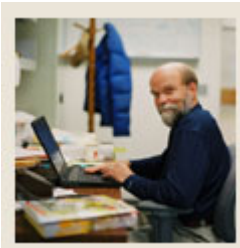
Create a PL/SQL script that selects the pidm and birth date from SWBPERS into a PL/SQL table. Sort by birth date.

Using the PL/SQL table, display the pidm and birth date when the birth date is equal to or greater than '01-JAN-1970', evaluating each record one at a time. Once a record's information is displayed, remove the record from the PL/SQL Table (**NOT the SWBPERS table**).

Exercise 6

Using the script from the previous exercise, display the pidm and birth date for all records where the birth date is less than '01-JAN-1970'. You should not have to reevaluate the birth date condition, because records whose birth dates were equal to or greater than '01-JAN-1970' were deleted in the previous step.

Note: Be sure to make use of some PL/SQL table attributes.



Section H: Procedures and Functions

Lesson: Overview

Jump to TOC

Introduction

The PL/SQL code we have seen so far could be used instead of a C or Cobol program. However, if we merely stop there, we are not taking advantage of all the power PL/SQL has to offer.

In this section, we are going to expand upon the PL/SQL layer by creating stored database subprograms that can be called from multiple environments.

Objectives

This section will examine the following:

- Advantages of modularizing code
- Differences between a function and a procedure
- Advantages of creating functions and procedures
- Create both a function and a procedure in the section exercises

Section contents

Section H: Procedures and Functions

Lesson: Modular Code

Jump to TOC

Modularizing code

Modularizing code is the process of breaking large processes down into simpler blocks of code.

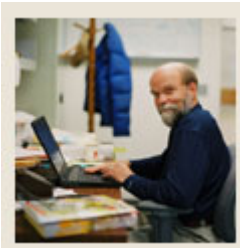
Advantages

- More reusable
- More manageable
- More readable
- More reliable

Anonymous Block

An anonymous block is a PL/SQL block that has no name. This is the type of block that we have been using so far – scripts of PL/SQL code that will be compiled and evaluated every time the script is run.

One anonymous block cannot call another anonymous block. Therefore, even if you modularize these blocks it doesn't give you the advantages listed above. To do that you need to create named PL/SQL blocks.



Section H: Procedures and Functions

Lesson: Layers of Oracle Programming

Jump to TOC

Layers

The most basic layer in which you interact with the database is SQL.

The next layer is the core of PL/SQL and the Oracle database which is comprised of stored PL/SQL units. Two of the primary PL/SQL objects needed to run PL/SQL are STANDARD and DBMS_STANDARD packages. Since these packages are created by default when you create a database and are integrated in the core of Oracle's underlying code, you don't have to explicitly reference them.

You can create entire applications with just these layers. However, your programs can be written much more efficiently by writing subprograms, which can be called from multiple applications.

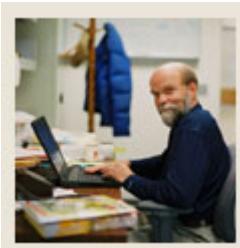
Anonymous Block

An anonymous block is a PL/SQL block that has no name. This is the type of block that we have been using so far – scripts of PL/SQL code that will be compiled and evaluated every time the script is run.

One anonymous block cannot call another anonymous block. Therefore, even if you modularize these blocks it doesn't give you the advantages listed above. To do that you need to create named PL/SQL blocks, also known as Stored Subprograms.

Stored Subprograms

Stored subprograms are named PL/SQL that is stored within the database. They can then be called from any application that allows the use of SQL or PL/SQL. There are two main types of subprograms - procedures and functions.



Section H: Procedures and Functions

Lesson: Procedure

Jump to TOC

Definition

A procedure is a stored subprogram that performs a specific action. By default, it returns no value.

Syntax

```
CREATE [OR REPLACE] PROCEDURE name [(parameter [, parameter, ...])]
IS
    [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

Parameter declarations are optional. However, when used they take on the following syntax:

```
parameter_name [IN | OUT | IN OUT] datatype [{:= | DEFAULT} expr]
```

PL/SQL has no explicit limit of parameters.

Notes

The keyword DECLARE is not used. Local declarations come after the keyword IS and before the keyword BEGIN.

A procedure has two parts - the specification and the body.

- The procedure specification begins with the keywords CREATE OR REPLACE PROCEDURE and ends with the procedure name or parameter list
- The procedure body begins with the keyword IS and ends with the keyword END, followed by an optional procedure name

Effects

- The procedure will be owned by a schema (username that created the procedure)
- The procedure is stored as compiled code
- The procedure can be called from multiple applications, such as another PL/SQL program, Oracle Forms, Pro*C, Pro*COBOL

Parameter Constraints

When a parameter is passed into a procedure, the constraints are passed also. Therefore, in a procedure declaration, it is illegal to constrain a CHAR or VARCHAR2 variable with a length and a numeric variable with a precision and scale.

For example, NUMBER(4) is illegal – use just the type declaration NUMBER.

You can use existing database column types when creating parameters by using the same %TYPE operator used to declare variables. The parameter takes on the datatype of the referenced column.

Examples

```
CREATE OR REPLACE PROCEDURE my_proc (pi_pidm NUMBER,
                                     pi_last_name VARCHAR2) IS
BEGIN
…
END;
CREATE OR REPLACE PROCEDURE my_proc2 (pi_pidm
                  swriden.swriden_pidm%TYPE,
                   pi_last_name  swriden.swriden_last_name%TYPE) IS

BEGIN
…
END;
```

Warning

There are known exploits of PL/SQL that use the lack of strong variable typing in parameters to a disadvantage. Therefore, it is recommended that parameters be assigned to local variables inside the code of the procedure using strongly typed data types.

```
CREATE OR REPLACE PROCEDURE my_proc (pi_pidm NUMBER,
                                     pi_text VARCHAR2) IS
     vi_pidm    NUMBER(4);
     vi_text    VARCHAR2(10);
BEGIN
  vi_pidm := pi_pidm;  -- error if value passed > 4 digits
  vi_text := substr(pi_text,1,10);  --ensures max of 10 char passed
…
END;
```

Section H: Procedures and Functions

Lesson: Parameters (Continued)

Jump to TOC

Modes

If a mode is not explicitly defined within the subprogram, the default of IN is assumed.

Mode	Description
IN	The value of the parameter is passed into the subprogram, and is considered read-only.
OUT	The value of the parameter being passed is ignored (write-only). Instead, the value is derived from within the subprogram, and the content of the formal parameter is assigned to the actual parameter and returned to the calling program.
IN OUT	A combination of both IN and OUT. The value of the parameter can be passed in, and the parameter value can be reassigned within the subprogram.

Default Value

```
Parameter_name [ mode ] parameter_type { := DEFAULT } initial_value;
```

Try to make the default parameters last in the list. This makes the subprogram easiest to call especially if the default value is being used. If the parameter is omitted, it will take on the default value.

Section H: Procedures and Functions

Lesson: Executing Procedures

Jump to TOC

Execution

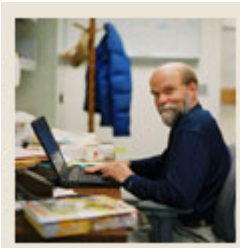
From within SQL*Plus, a procedure can be executed by using the EXECUTE command followed by the procedure name.

Any arguments required by the procedure must be passed in parentheses. Be sure to enclose string and date values in single quotes.

```
SQL> EXECUTE procedure_name(parm1, parm2...);
SQL> EXECUTE sp_my_procedure(12345,'Smith', '21-Dec-2007');
```

When executing a procedure through PL/SQL you can omit the keyword EXECUTE.

```
BEGIN
      sp_my_procedure(12345,'Smith', '21-Dec-2007');
END;
/
```



Example

```
/* Purging block. Retrieves non-current row from SWRIDEN based on the
pidm passed in. It inserts the row into SWRIDEN_HISTORY.  After the
row has been inserted, then the row from SWRIDEN is deleted. */

CREATE OR REPLACE PROCEDURE purge_one_swriden (pi_pidm NUMBER) IS
    lv_sqlcode     NUMBER;
    lv_sqlerrm     VARCHAR2(55);
   swriden_row swriden%ROWTYPE;
   CURSOR c1 IS
      SELECT *
        FROM swriden
       WHERE swriden_change_ind IS NOT NULL
          AND swriden_pidm = pi_pidm
          FOR UPDATE;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO swriden_row;
      EXIT WHEN c1%NOTFOUND;
         INSERT INTO swriden_history (swriden_hist_pidm,
         swriden_hist_id, swriden_hist_last_name,
         swriden_hist_first_name, swriden_hist_change_ind,
         swriden_hist_activity_date)
          VALUES (swriden_row.swriden_pidm, swriden_row.swriden_id,
                  swriden_row.swriden_last_name,
                  swriden_row.swriden_first_name,
                  swriden_row.swriden_change_ind, SYSDATE);
      DELETE FROM swriden
         WHERE CURRENT OF c1;
   END LOOP;
   COMMIT;
  EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK;
      lv_sqlcode := SQLCODE;
      lv_sqlerrm := SUBSTR(SQLERRM, 1, 55);
      BEGIN
         DBMS_OUTPUT.PUT_LINE(lv_sqlcode||' '||lv_sqlerrm);
      END;

END purge_one_swriden;
```

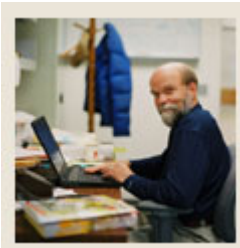
Actual Parameter vs. Formal Parameter

You can also pass parameters to procedures in the form of variables, also known as the actual parameter. The variable name passed to the procedure need not match the name of the parameter as defined in the procedure, the formal parameter.

In the example of purge_one_swriden, the subprogram is looking for a pidm to be passed in. Below is an example of how the procedure might be called from PL/SQL.

```
  DECLARE
     CURSOR purge_cursor IS
     SELECT swbpers_pidm
       FROM swbpers
      WHERE swbpers_pidm=12340;
  BEGIN
     FOR purge_rec IN purge_cursor LOOP
        purge_one_swriden(purge_rec.swbpers_pidm);
     END LOOP;
  END;
/
```

In the example above the actual parameter is the *purge_rec.pidm* variable, since this is the variable being passed into the procedure. When control passes to the subprogram, the value is assigned to *pi_pidm*. In this case, *pi_pidm* is the formal parameter, or the parameter name defined within the subprogram.



Section H: Procedures and Functions

Lesson: Positional vs. Named Notation

Jump to TOC

Notation types

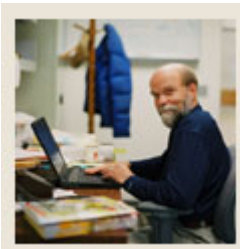
The previous examples have all used positional notation. Positional notation assigns the variables to parameters by the order in which they are passed.

You may also assign variables using named notation.

```
  DECLARE
     Variable_one   VARCHAR2(2);
     Variable_two   VARCHAR2(5);
     Variable_three VARCHAR2(1);
  BEGIN
     call_function(Parameter_two => Variable_two,
                   Parameter_one => Variable_one,
                   Parameter_three => Variable_three);
  …
  END;
/
```

Notes:

- Named Notation allows you to omit parameters that have default values.
- Named Notation allows for flexibility in modifying parameters – if the stored program changes the order of parameters the calling program need not be modified.
- Neither positional nor named notation is more efficient, therefore it is based upon preference.



Section H: Procedures and Functions

Lesson: Functions

Jump to TOC

Definition

A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions **must** return a variable using the RETURN clause.

Syntax

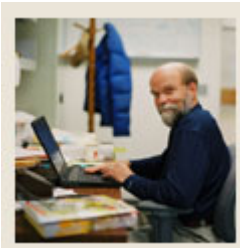
```
CREATE [ OR REPLACE ] FUNCTION name [(parameter [, parameter, ...])]
    RETURN datatype IS
     [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

Parameters are optional but when used take on the same syntax, have the same properties and obey the same rules as parameters in procedures:

```
parameter_name [IN | OUT | IN OUT] datatype [{:= | DEFAULT} expr]
```

Example

```
/* Accepts an id, and returns the PIDM. */
CREATE OR REPLACE FUNCTION get_pidm (pi_id VARCHAR2) RETURN NUMBER IS
  lv_pidm swriden.swriden_pidm%TYPE;
BEGIN
   SELECT swriden_pidm
     INTO lv_pidm
     FROM swriden
    WHERE swriden_id = pi_id;
   RETURN lv_pidm;
END get_pidm;
```



Notes

The keyword DECLARE is not used. Local declarations are placed after the keyword IS and before the keyword BEGIN.

Like a procedure, a function has two parts - the specification and the body.

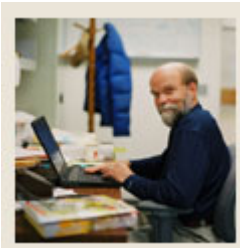
- The function specification begins with the keywords CREATE OR REPLACE FUNCTION and ends with a RETURN clause.
- The function body begins with the keyword IS and ends with the keyword END followed by an optional function name.

The RETURN statement immediately completes the executions of a subprogram, whether it is the last statement or not.

RETURN statements

A good programming practice is to only have one RETURN statement per function to increase maintainability and make the code easier to follow.

```
   /* Example of multiple return statements */

   CREATE OR REPLACE FUNCTION get_pidm (pi_id VARCHAR2)
          RETURN VARCHAR2 IS
     lv_count    NUMBER;
   BEGIN
      SELECT count(*)
        INTO lv_count
        FROM swriden
       WHERE swriden_id = pi_id;
       IF lv_count = 1 THEN
          RETURN 'ID only has one record';
       ELSE
          RETURN 'ID has multiple records';
       END IF;
   END get_pidm;


   /* Example using single return statement – easier to maintain and
follow */
   CREATE OR REPLACE FUNCTION get_pidm (pi_id VARCHAR2)
          RETURN VARCHAR2 IS
     lv_count    NUMBER;
     lv_message  VARCHAR2(50);
   BEGIN
      SELECT count(*)
        INTO lv_count
        FROM swriden
       WHERE swriden_id = pi_id;
       IF lv_count = 1 THEN
          lv_message := 'ID only has one record';
       ELSE
          lv_message := 'ID has multiple records';
       END IF;
       RETURN lv_message;
   END get_pidm;
```



Section H: Procedures and Functions

Lesson: Calling a Function

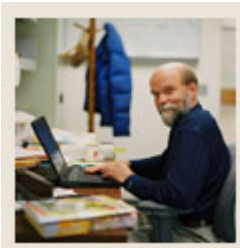
Jump to TOC

Calling a Function

Because a function returns a value there needs to be a way of storing that return value.

When the function is called from PL/SQL the return value can be assigned to a variable. If the function is called in a SELECT, INSERT, UPDATE, or DELETE statement, it is treated as if it were a database column.

```
DECLARE
     lv_pidm swraddr.swraddr_pidm%TYPE;
BEGIN
   …
   lv_pidm := get_pidm('123G');
   …
END;

OR

SELECT get_pidm('123G')
  FROM dual;
```



Section H: Procedures and Functions

Lesson: What Can Functions Do For You?

Jump to TOC

Benefits

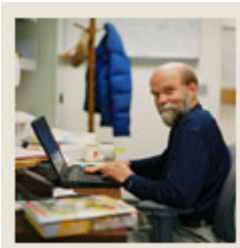
You can create a library of customized calculations.

Functions can also be called within a SELECT statement.

```
SELECT get_id(swbpers_pidm),swbpers_ssn, …
  FROM swbpers;
```

You don't need a join! Any PRO*C, PRO*COBOL, or Oracle Form that needs an ID can use this function, without explicitly referring to SWRIDEN.

Banner contains a large library of functions to perform common actions like calculate age based on a birthdate and some date in time (F_CALCULATE_AGE) and figure out which level a student is at (F_CLASS_CALC_FNC). Check for an existing Banner function before you re-invent you own.



Section H: Procedures and Functions

Lesson: Example Function

Jump to TOC

Example

```
/* Accepts the parameter of pidm_in and returns the age calculated
from SWBPERS where pidm is equal to pidm_in. If no row is found for
the pidm, then a message is returned to indicate this.  If any other
error occurs, then the error number and error message are returned.
*/

CREATE OR REPLACE FUNCTION get_age (pi_pidm IN NUMBER)
       RETURN VARCHAR2 IS
  lv_age NUMBER(3);
BEGIN
  SELECT TRUNC(MONTHS_BETWEEN(SYSDATE,swbpers_birth_date)/12)
    INTO lv_age
    FROM swbpers
   WHERE swbpers_pidm = pi_pidm;
  RETURN TO_CHAR(LV_AGE);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      RETURN 'No data in SWBPERS';
   WHEN OTHERS THEN
      DECLARE
         lv_sqlcode NUMBER(5);
         lv_sqlerrm VARCHAR2(30);
      BEGIN
         lv_sqlcode := SQLCODE;
         lv_sqlerrm := SUBSTR(SQLERRM,1,30);
         RETURN lv_sqlcode || lv_sqlerrm;
      END;
END GET_AGE;
/
```

Execute the function

To execute the function, we can do the following:

```
SQL> SELECT get_age(12340) FROM DUAL;

GET_AGE(12340)
----------------------
32
```

Most likely, we will use the function within a SELECT statement that returns more than just the age:

```
SQL> SELECT swriden_first_name, swriden_last_name,
            get_age(swriden_pidm)
       FROM swriden
      WHERE swriden_change_ind IS NULL;


FIRST_NAME      LAST_NAME      GET_AGE(PIDM)
--------------- ------------------ -----------------
Julie           Brown                27
Robert          Smith                29
Peter           Johnson              No data in SWBPERS
```

The function can also be called from a stored or anonymous PL/SQL block:

```
SQL>  declare
  2     lv_age  number;
  3  begin
  4     lv_age := get_age(12340);
  5     DBMS_OUTPUT.PUT_LINE('Age is: ' || lv_age);
  6* end;
SQL> /
Age is: 33

PL/SQL procedure successfully completed.
```

Compilation errors

If there were compilation errors for functions or procedures you will receive a message saying your PL/SQL had errors:

```
SQL>  CREATE OR REPLACE FUNCTION get_pidm (pi_id VARCHAR2)
 2         RETURN VARCHAR2 IS
 3    lv_count    NUMBER;
 4    lv_message  VARCHAR2(50);
 5  BEGIN
 6     SELECT count(*)
 7       INTO lv_count
 8       FROM swriden
 9      WHERE swriden_id = pi_id;
10      IF lv_count = 1 THEN
11         lv_message := 'ID only has one record';
12      ELSE
13         lv_mvessage := 'ID has multiple records';
14      END IF;
15      RETURN lv_message;
16* END get_pidm;
SQL> /

Warning: Function created with compilation errors.
```

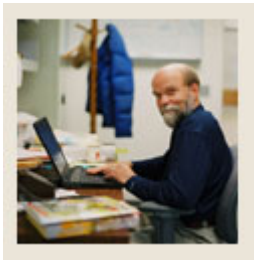
The code is still stored in the database but the procedure or function is marked INVALID and cannot be called until it compiles completely.

To find out what the errors are type the following:

```
SQL> show errors
Errors for FUNCTION GET_PIDM:

LINE/COL ERROR
-------- -----------------------------------------------------------
13/8     PLS-00201: identifier 'LV_MVESSAGE' must be declared
13/8     PL/SQL: Statement ignored
```

You cannot easily fix the stored procedure or function directly when errors occur. You should store your source code in a text file that can be modified to correct any errors and then re-run to store the new code in the database.

The REPLACE option of the CREATE or REPLACE command will allow the existing stored database procedure/function to be replaced with the new version.

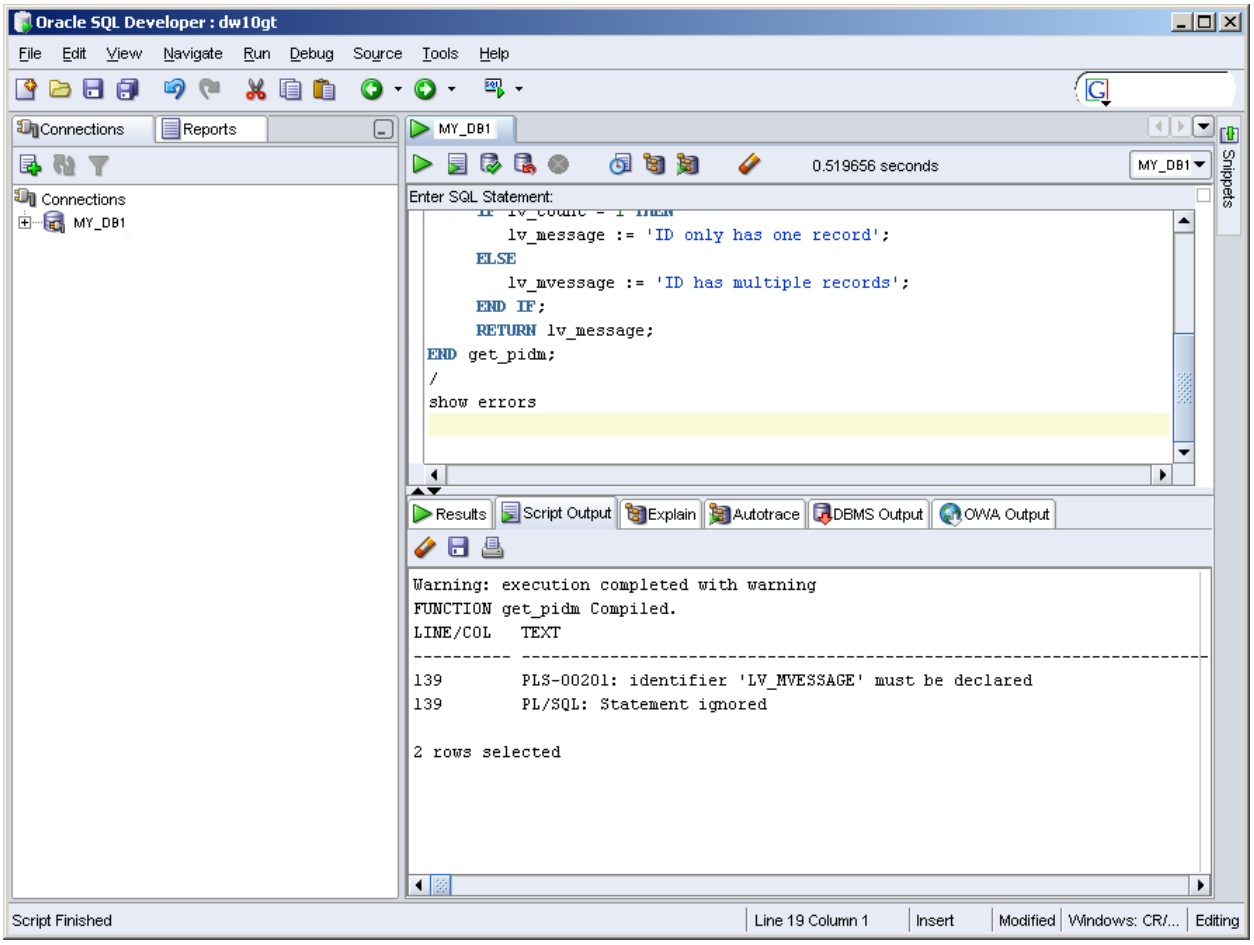


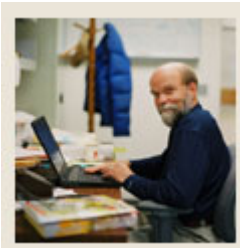
Section H: Procedures and Functions

Lesson: Handling Compilation Errors (continued)

Jump to TOC

Handling Errors in SQL Developer





Dictionary Views

There are a number of data dictionary views that can be used to view your PL/SQL objects.

USER_OBJECTS – shows all objects owned by the user including tables and PL/SQL
USER_SOURCE – contains the source code, line by line of all user owned PL/SQL
USER_DEPENDENCIES – contains the related objects called in stored PL/SQL

Retrieve Code

If you ever lose a copy of your stored PL/SQL code it can be retrieved from the USER_SOURCE view. Each line of your code is stored as a row in this view.

```
SQL> desc user_source
 Name                      Null?    Type
 ------------------------- -------- ----------------------------
 NAME                               VARCHAR2(30)
 TYPE                               VARCHAR2(12)
 LINE                               NUMBER
 TEXT                               VARCHAR2(4000)
```

Validity

To check whether a subprogram is valid, query the data dictionary view USER_OBJECTS.

```
SQL> SELECT OBJECT_NAME, STATUS
       FROM ALL_OBJECTS
      WHERE OBJECT_NAME = 'GET_ID';
```

Dependencies

As a subprogram is created in the database, it is compiled. During the compilation, Oracle keeps track of the dependencies such as which tables or other objects the subprogram is referencing. A change to one of the dependencies later on, such as altering or dropping a table, could potentially affect your subprogram to make it invalid.

Referenced Objects

To find which subprograms will be affected if you change a table check the USER_DEPENDENCIES view:

```
SQL> SELECT name, type, referenced_owner, referenced_name
  2  FROM user_dependencies
  3  WHERE referenced_name = 'SWRIDEN';

NAME                 TYPE      REFERENCED_OWNER REFERENCED_NAME
-------------------- --------- ---------------- ------------------
F_JOB_INFO           FUNCTION  SATURN           SWRIDEN
F_PLEDGE_SOLICITOR   FUNCTION  SATURN           SWRIDEN
F_ORDER_BY           FUNCTION  SATURN           SWRIDEN
F_FACT_NAME          FUNCTION  SATURN           SWRIDEN
F_INTERVIEWER_NAME   FUNCTION  SATURN           SWRIDEN
F_SUPERVISOR_INFO    FUNCTION  SATURN           SWRIDEN
```

Query the data dictionary view DBA_OBJECTS for a list of all objects that may be invalid as a result of changes to referenced or referencing objects.

```
SQL> SELECT OWNER, OBJECT_NAME, STATUS FROM DBA_OBJECTS
  2  WHERE STATUS <> 'VALID';
```

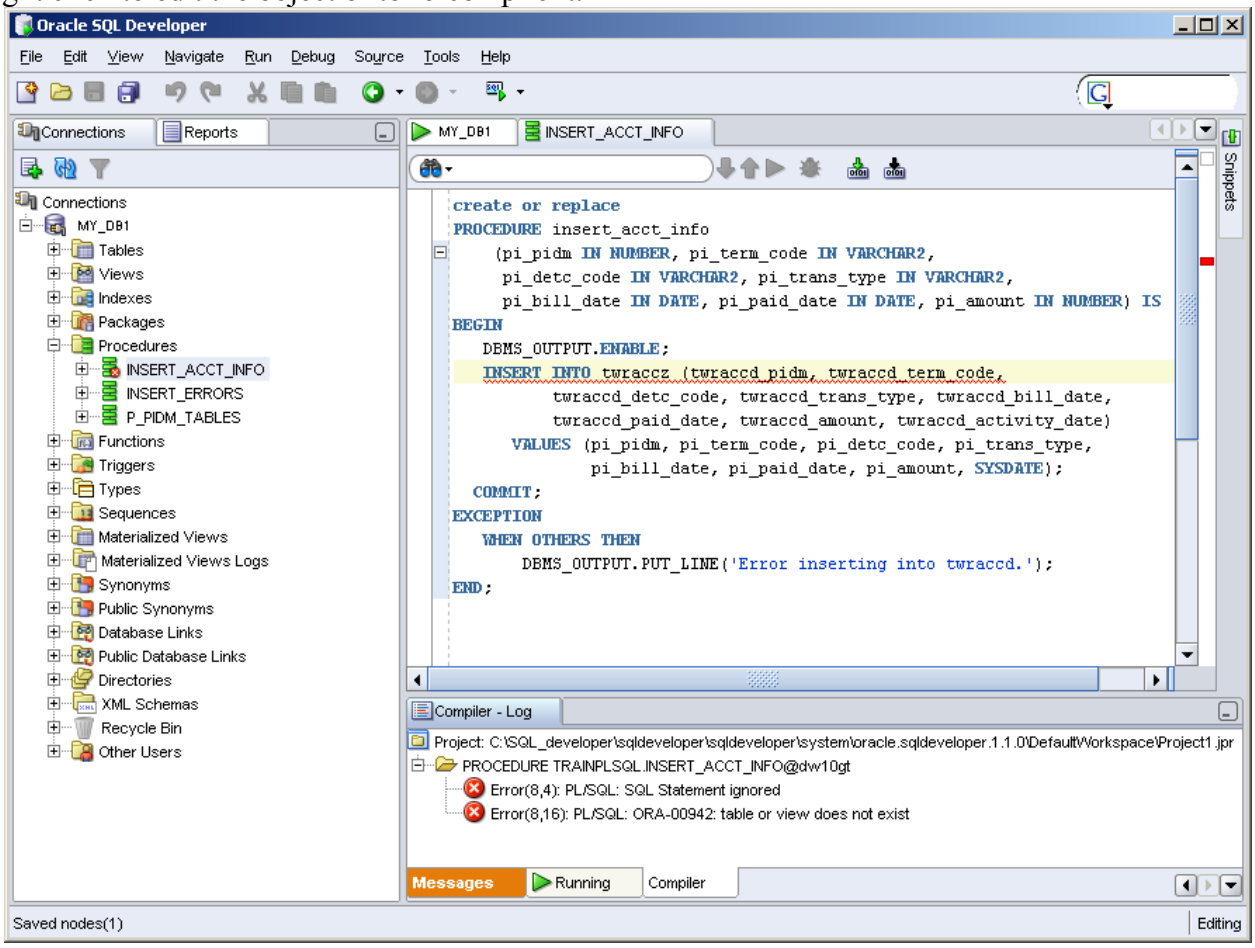
Section H: Procedures and Functions

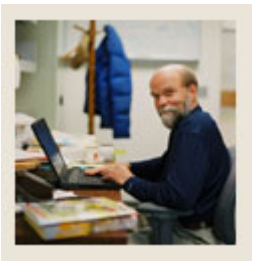
Lesson: Locate Objects in the Database (continued)

Jump to TOC

In SQL Developer, locate your object through the Heirarchical Tree. Only the objects you own are in this tree. If you need to view objects owned by other users, click on the Other Users option at the bottom of the tree.

Invalid objects will have a red 'x' next to their name. Click on the object to view the errors. Right click to edit the object or to re-compile it.





Section H: Procedures and Functions

Lesson: Remove Subprograms

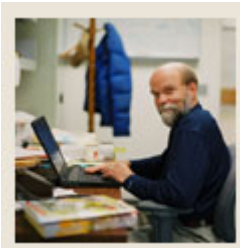
Jump to TOC

Removing Procedures and Functions

If you want to remove a subprogram from the database use the DROP command.

```
DROP PROCEDURE <procedure_name>;

DROP FUNCTION <function_name>;
```

Once removed, the objects can no longer be referenced. If the object was called from another sub-program it will cause that sub-program to go invalid.



Section H: Procedures and Functions

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

In the account table TWRACCD, an amount is considered unpaid if the PAID_DATE is null. Create a stored function called calc_amt_owed, which returns the sum amount of unpaid bills when a pidm is passed.

Exercise 2

Select the first name, last name, and amount owed, using the function you just created.

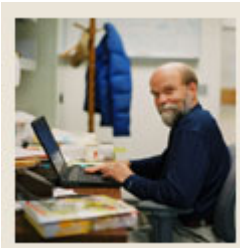
Exercise 3

Create a procedure called insert_acct_info that inserts an account transaction into the account table (TWRACCD). The procedure should have parameters for:

- PIDM
- TERM_CODE
- DETC_CODE
- TRANS_TYPE
- BILL_DATE
- PAID_DATE
- AMOUNT

ACTIVITY_DATE should be automatically derived within the procedure.

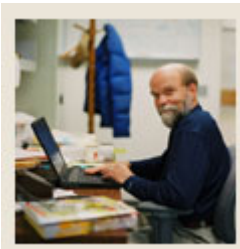
Be sure to test your procedure by calling the procedure to insert a record.



Exercise 4

Create a procedure that inserts into the TEMP table when an error occurs in a script (**replacing the bolded code below**). The procedure should pass in the error code and error message.

```
  ...
WHEN OTHERS THEN
            lv_sqlcode := SQLCODE;
            lv_sqlerrm := SUBSTR(SQLERRM, 1, 55);
            ROLLBACK;
            INSERT INTO temp (col1, col2, message)
            VALUES (lv_sqlcode, sysdate, lv_sqlerrm);
            COMMIT;
```

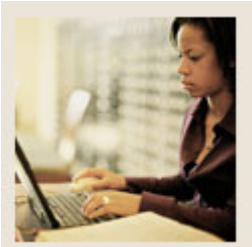


Exercise 5

Write a quick PL/SQL script that causes an error, such as retrieving multiple rows into a SELECT INTO statement, and calls your procedure in Exercise 4 when it encounters the error. Select from the TEMP table to make sure that the error handler is working properly.

Exercise 6

Locate your newly created PL/SQL objects in the database using USER_OBJECTS, USER_SOURCE, and USER_DEPENDENCIES.



Section I: Packages

Lesson: Overview

Jump to TOC

Introduction

As you create stored functions and procedures, you will be able to reduce the amount of code that is in each application. However, when you begin to rely heavily upon stored subprograms, you will soon realize that it will be difficult to know what each subprogram is used for. Of course, you will want to add comments in each subprogram. But beyond this, what can you do? You can bundle subprograms into packages.

Not only will packages allow us to bundle subprograms, but also global cursors and variables.

Objectives

By the end of this section, each attendee will be able to

- create database packages
- list the security benefits that packages offer
- list some built-in packages that Oracle provides.

Section contents

Section I: Packages

Lesson: Benefits of Packages

Jump to TOC

Packages promote the object-oriented model

Although packages do not support every concept in the object-oriented design model, they do provide for some principles such as encapsulation. Packages hide the implementation so that only the package (not your application) is affected if the implementation changes

The Oracle RDBMS automatically tracks the validity of all program objects (procedures, functions, and packages) stored in the database. It determines what other objects that program is dependent on, such as other packages. If a dependent object changes, then all programs that rely on that object are flagged invalid. The dependent objects are automatically re-compiled the next time they are called.

Performance Benefits

When one object in a package is referenced for the first time, the entire package (already compiled and validated) is loaded into the Shared Global Area (SGA) of the database. All other package elements are thereby made immediately available for future calls to the package. PL/SQL does not have to keep retrieving program elements from disk each time a new object is referenced.

In a distributed environment where packages are executed across a local area network, minimization of network traffic can boost performance.

Package Security

As a PL/SQL object, security on the package is limited to the same principles as the other objects; grant privileges, object owners, etc. However, by nature, the package can secure the objects it contains thereby hiding some of the most detailed aspects of your programs.

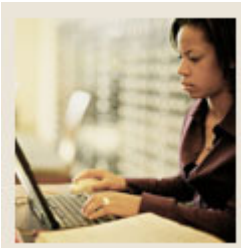
When you build a package, you decide which of the elements are public (referenced outside the package) and which are private (available only within the package itself) using the package specification and the package body. Public objects are defined within the package specification. Private objects are defined within the package body. So, by defining an object as private, you can essentially protect the most confidential of business rules.

Easier Application Design

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package body fully until you are ready to complete the application

Added Functionality

Packaged public variables and cursors persist for the duration of a session. So, they can be shared by all subprograms that execute in the environment. Also, they allow you to maintain data across transactions in that session without having to store it in the database



Section I: Packages

Lesson: Package Structure

Jump to TOC

Package components

The package consists of two distinct structures:

- **The specification or header**
  Specification (**spec** for short) is the public interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use
- **The body**
  The package body implements the package spec. That is, the package body contains the implementation of every cursor and subprogram declared in the package spec

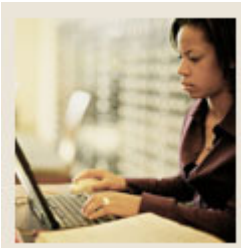
  In functions and procedures, the keyword IS connects the specification to the body of the subprogram. For a package, the specification and the body are not connected - they are separate, distinct code structures.

Package Specification

To bundle PL/SQL objects together, declare which components of your package are available to other applications. In other words, which objects are public, what type of objects they are, and what parameters are expected when a program unit in the package is called?

```
CREATE OR REPLACE PACKAGE package_name
IS
[ declarations of public variables and types ]
[ specifications of public cursors ]
[ specifications of modules (i.e. functions and procedures) ]
END [ package_name ];
```

```
CREATE OR REPLACE package general_person IS
   gv_ACTIVITY      date := SYSDATE;
   gv_EXCEPTION     EXCEPTION;
   CURSOR id_cursor (pi_pidm IN NUMBER)
      RETURN swriden%ROWTYPE;
   CURSOR get_addresses(pi_pidm NUMBER) IS
      SELECT swraddr_atyp_code,swraddr_street_line1,
             swraddr_stat_code, swraddr_zip
        FROM swraddr
       WHERE swraddr_pidm = pi_pidm;
   FUNCTION get_id(pi_pidm IN NUMBER) RETURN VARCHAR2;
END general_person;
/
```



Package Body

Once the specification declares to the database what to expect from your package, you can decide whether you need to code a package body. Does your specification declare any hidden cursors? Does your specification declare any functions or procedures? If so, then there is PL/SQL code which has to be assigned to those objects before they are complete.

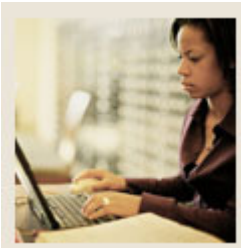
The role of the package body is to contain the code behind those objects that require the specific PL/SQL language constructs.

```
CREATE OR REPLACE PACKAGE BODY package_name
IS
[ declarations of private variables and types ]
[ specifications of private cursors ]
[ specifications of private modules (i.e. functions and procedures) ]

[ BEGIN
   executable statements ]
[ EXCEPTION
   exception statements]
END [ package_name ];
```

Package Body

```
CREATE OR REPLACE PACKAGE BODY GENERAL_PERSON
IS

   CURSOR id_cursor(pi_pidm IN NUMBER) RETURN swriden%ROWTYPE IS
      SELECT *
        FROM swriden
       WHERE swriden_pidm = pi_pidm;

   FUNCTION get_id(pi_pidm IN NUMBER) RETURN VARCHAR2 IS
      lv_id swriden.swriden_id%TYPE;
   BEGIN
      SELECT swriden_id
        INTO lv_id
        FROM swriden
       WHERE swriden_change_ind IS NULL
         AND swriden_pidm = pi_pidm;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         RETURN 'No rows found.';
      WHEN TOO_MANY_ROWS THEN
         RETURN 'Too many rows.';
      WHEN OTHERS THEN
         DECLARE
            lv_err_msg VARCHAR2(200) := SUBSTR(SQLERRM,1,200);
         BEGIN
            RETURN lv_err_msg;
         END;
   END get_id;

END general_person;
/
```

Note: You cannot compile a package body without a package specification.



Package Elements - Syntax

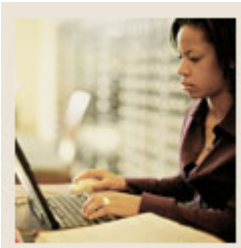
To reference one of the public variables in a Package Specificaton use the syntax *package_name.variable_name*

```
IF general_person.gv_activity > '01-JAN-07' THEN
   …
END IF;
```

Cursors - Syntax

To reference one of the public cursors in a Package Specificaton use the syntax *package_name.cursor_name*

```
OPEN general_person.id_cursor(pi_pidm);
```



Section I: Packages

Lesson: Unqualified Package Elements

Jump to TOC

Unqualified elements

When a public package object is referenced within another object in the same package, it does not require the package name preceding it.

```
INSERT INTO SWRIDEN
   (___,____,____, swriden_activity_date)
VALUES (____,____, gv_activity);
```

Data declared at the package level persist for as long as the session is active. However, each Oracle session has its own private PL/SQL area. This means that if multiple people are calling the same package, they will have their own set of global variables, cursors, etc.

Public vs. private data

Allows you to enforce information hiding.

```
CREATE OR REPLACE PACKAGE general_person IS
            /* Public.  Any session can reference it. */
   gv_pidm NUMBER;
END;
/

CREATE OR REPLACE PACKAGE BODY general_person IS
           /* Private. */
   lv_id VARCHAR2(9);
END general_person;
/
```

Test Cases

```
DECLARE
   lv_pidm NUMBER(8);
BEGIN
   lv_pidm := general_person.gv_pidm;  /* Legal. */
END;

DECLARE
   lv_id VARCHAR2(9);
BEGIN
   lv_id := general_person.lv_id;  /* Illegal. lv_id is PRIVATE */
END;
/
```

Package specification

```
CREATE OR REPLACE PACKAGE registration_lib
IS
  FUNCTION calc_credit_hours (pi_pidm IN swriden.swriden_pidm%TYPE,
                              pi_term IN swvterm.swvterm_term_code%TYPE)
     RETURN NUMBER;

  PROCEDURE display_ticket (pi_pidm IN swriden.swriden_pidm%TYPE,
                            pi_term IN swvterm.swvterm_term_code%TYPE);
END registration_lib;
```

The package specification declares a package called registration_lib. This package groups together two modules:

- calc_credit_hours function
- display_ticket procedure

These two modules specifically relate to registration data.

By examining the specification, another programmer can tell exactly how to reference the objects and what to pass to them.

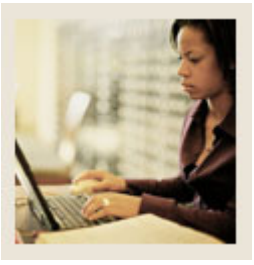
Package body

```
CREATE OR REPLACE PACKAGE BODY registration_lib
IS
   FUNCTION calc_credit_hours (pi_pidm IN swriden.swriden_pidm%TYPE,
                               pi_term IN swvterm.swvterm_term_code%TYPE
                              )
      RETURN NUMBER
   IS
      lv_total_hours   NUMBER (6, 2);
      CURSOR swrregs_cur (pi_pidm IN swriden.swriden_pidm%TYPE,
                          pi_term IN swvterm.swvterm_term_code%TYPE
                         )
        IS
           SELECT swrregs_crn, swrregs_gpa
             FROM swrregs
            WHERE swrregs.swrregs_pidm = pi_pidm AND
                  swrregs.swrregs_term_code = pi_term;
      CURSOR swvcrse_cur (pi_crn IN swvcrse.swvcrse_crn%TYPE)
        IS
           SELECT swvcrse_credit_hours
             FROM swvcrse
            WHERE swvcrse.swvcrse_crn = pi_crn;
      swvcrse_rec   swvcrse_cur%ROWTYPE;
   BEGIN
      lv_total_hours := 0;
      FOR swrregs_rec IN swrregs_cur (pi_pidm, pi_term)
      LOOP
         OPEN swvcrse_cur (swrregs_rec.swrregs_crn);
         FETCH swvcrse_cur INTO swvcrse_rec;
         IF swvcrse_cur%FOUND
         THEN
            lv_total_hours := lv_total_hours +
                              swvcrse_rec.swvcrse_credit_hours;
         END IF;
         CLOSE swvcrse_cur;
      END LOOP;
      RETURN lv_total_hours;
   END calc_credit_hours;
```

(continued….)

Example code, continued

```
PROCEDURE display_ticket (pi_pidm IN swriden.swriden_pidm%TYPE,
                          pi_term IN swvterm.swvterm_term_code%TYPE
                         )
 IS
  CURSOR swrregs_cur (pi_pidm IN swriden.swriden_pidm%TYPE,
                      pi_term IN swvterm.swvterm_term_code%TYPE
                     )
  IS
     SELECT swrregs_crn, swrregs_gpa, swvcrse_desc
       FROM swrregs, swvcrse
      WHERE swrregs_pidm = pi_pidm
        AND swrregs_term_code = pi_term
        AND swvcrse_crn = swrregs_crn;
BEGIN
 DBMS_OUTPUT.put_line ('CRN    DESCRIPTION                    GPA'  );
 DBMS_OUTPUT.put_line ('------ ------------------------------ -----');
 FOR swrregs_rec IN swrregs_cur (pi_pidm, pi_term)
   LOOP
      EXIT WHEN swrregs_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE (
        RPAD(swrregs_rec.swrregs_crn, 7, ' ' ) ||
        RPAD(swrregs_rec.swvcrse_desc, 31, ' ' ) ||
        swrregs_rec.swrregs_gpa);
   END LOOP;
 END display_ticket;
END registration_lib;
/
```



Section I: Packages

Lesson: Synchronize the Specification and the Body

Jump to TOC

Synchronization

It is imperative that you keep the package specification synchronized with the body and vice versa. If you do not, the compiler will generate the following error:

```
PLS-00232:  subprogram 'name' is declared in a package specification
and must be defined in the package body
```

Common causes of this error include:

- A parameter was added to a procedure or function in the body that has not been added to the specification
- A parameter was added to a procedure or function in the specification that has not been added to the body
- The datatype of a parameter changed and now the specification does not match the body
- The name of a parameter changed and now the specification does not match the body
- A new function or procedure was added to the body but the public declaration has not been added to the specification
- The name of a procedure or function was changed and the specification no longer matches the body

Problems with redefinition

In the previous example, notice the use of the swrregs_cur cursor. In both the function and the procedure, the cursor performs the same action; it loads the student registration records for a given term. If you think about it, there might be a need for several different types of applications to have access to these records. Do we really want to redefine this cursor repeatedly; taking the chance that one module will declare the subset of records differently than another?

Public cursor element

Because we have declared the cursors within each of the modules, the cursors are considered private, meaning that they cannot be referenced anywhere but within the package. This is a nice feature in some situations, but in this instance, we really want the cursor to be public. Consider the following modifications to our package specification:

```
CREATE OR REPLACE PACKAGE registration_lib
IS
   CURSOR swrregs_cur (
      pi_pidm IN swriden.swriden_pidm%TYPE,
      pi_term IN swvterm.swvterm_term_code%TYPE
   )
   IS
      SELECT swrregs_crn, swrregs_gpa, swvcrse_desc
        FROM swrregs, swvcrse
       WHERE swrregs_pidm = pi_pidm
         AND swrregs_term_code = pi_term
         AND swvcrse_crn = swrregs_crn;
   FUNCTION calc_credit_hours
     (
      pi_pidm IN swriden.swriden_pidm%TYPE,
      pi_term IN swvterm.swvterm_term_code%TYPE
     )
      RETURN NUMBER;
   PROCEDURE display_ticket
     (
      pi_pidm IN swriden.swriden_pidm%TYPE,
      pi_term IN swvterm.swvterm_term_code%TYPE
     );
END registration_lib;
```

Cursor declaration

Notice that the cursor is now declared in the package specification, which means it no longer needs to be defined within the package body:

```
CREATE OR REPLACE PACKAGE BODY registration_lib
IS
    CURSOR swvcrse_cur(pi_crn IN swvcrse.swvcrse_crn%TYPE)
    IS
        SELECT  swvcrse_credit_hours
        FROM  swvcrse
        WHERE  swvcrse_crn = pi_crn;

   FUNCTION calc_credit_hours (pi_pidm IN swriden.swriden_pidm%TYPE,
                               pi_term IN swvterm.swvterm_term_code%TYPE)
      RETURN NUMBER
      IS
         lv_total_hours NUMBER(6,2);
         swvcrse_rec swvcrse_cur%ROWTYPE;

   BEGIN
        lv_total_hours := 0;
        FOR swrregs_rec IN swrregs_cur(pi_pidm,pi_term)
        LOOP
            OPEN swvcrse_cur(swrregs_rec.swrregs_crn);
            FETCH swvcrse_cur INTO swvcrse_rec;
            IF swvcrse_cur%FOUND THEN
               lv_total_hours := lv_total_hours +
                                 swvcrse_rec.swvcrse_credit_hours;
            END IF;
        CLOSE swvcrse_cur;
      END LOOP;
      RETURN lv_total_hours;
   END calc_credit_hours;
```

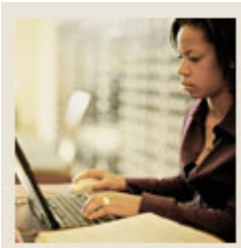
Cursor declaration (cont.)

```
    PROCEDURE display_ticket (pi_pidm IN swriden.swriden_pidm%TYPE,
                              pi_term IN swvterm.swvterm_term_code%TYPE)
    IS
    BEGIN
      dbms_output.put_line('CRN    DESCRIPTION                GPA');
      dbms_output.put_line('------ ------------------------- -----');
      FOR swrregs_rec in swrregs_cur(pi_pidm,pi_term)
      LOOP
          dbms_output.put_line (
          RPAD(swrregs_rec.swrregs_crn, 7, ' ' ) ||
          RPAD(swrregs_rec.swvcrse_desc, 26, ' ' ) ||
          swrregs_rec.swrregs_gpa);
      END LOOP;
    END display_ticket;

  END registration_lib;
/
```

Publically declared cursor

Now that the cursor has been declared as public, the modules can use one single format for retrieving the registration records. Likewise, if the cursor were to change, you would only have to change it in one place. Additionally, since the cursor is declared public, it is legal to reference the cursor within any other PL/SQL object external to this package. For instance, if you were going to build a transcript, you would want to repeat calls to this cursor.

```
FOR swvterm_rec in swvterm_cur
LOOP
    OPEN registration_lib.swrregs_cur
          (swrregs_pidm,swvterm_rec.swvterm_term_code);
    FETCH registration_lib.swrregs_cur2 INTO local_regis_rec;
    IF registration_lib.swrregs_cur%FOUND THEN
       dbms_output.put_line(local_regis_rec.swrregs_crn ||
                            local_regis_rec.swvcrse_desc ||
                            local_regis_rec.swrregs_gpa);
       lv_sem_gpa_pts := lv_sem_gpa_pts +
                         local_regis_rec.swrregs_gpa;
    END IF;
    CLOSE registration_lib.swrregs_cur2;
END LOOP;
```



Package body

Consider this, however; if you only had variables, constants, cursors and exception types declared in a package specification, would the package body be required? The answer is no, because the specification has declared to the database that there are no 'incomplete' objects within the package.

Variables, constants, cursors and exceptions are complete once they are defined within the declaration section of any PL/SQL block. By building such a package specification you can make a number of standard variables, constants, cursors and/or exceptions available for use throughout your applications simply by declaring them once within a package.

```
CREATE OR REPLACE PACKAGE registration_globals
IS
   gv_max_credit_hours  NUMBER(6,2) := 12;
END registration_globals;

IF lv_total_hours > registration_globals.gv_max_credit_hours THEN …
```

Remember the performance advantages of using packages? Once an object within a package is referenced, the entire package is loaded into the SGA so that future references to the package do not require any disk access. If you were to have such a package specification as described above, any 'global' variables, constants, cursors and/or exception types declared are instantly accessible as soon as the first reference to 'registration_globals' is made.

Section I: Packages

Lesson: Overloading Packages

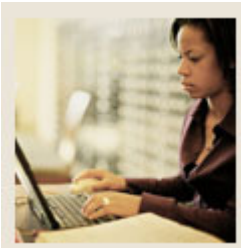
Jump to TOC

Definition

A package can be overloaded, which means that more than one procedure or function has the same name, but with different parameters. Based on the datatype or number of parameters, Oracle will be able to deduce which subprogram needs to run.

Example

```
CREATE OR REPLACE PACKAGE general_person IS
   PROCEDURE get_name (pi_pidm IN NUMBER);
   PROCEDURE get_name (pi_ID IN VARCHAR2);
END general_person;
```



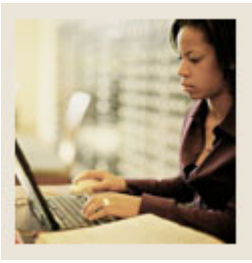
Section I: Packages

Lesson: Recommendations for Using Packages

Jump to TOC

Recommendations

- Do not build related functions and procedures standalone; instead, embed them into a package where appropriate (grouping related objects together in a package)
- Use consistent and effective coding style
- Overload for smart packages
- Make your programs case-insensitive
- Use packages and public elements to avoid duplication of coding and declarations



Subprogram Execution

The security for packages is the same as for Procedures and Functions. To be able to call a package, procedure, or function that is owned by another user, you have to be granted the EXECUTE privilege on that stored object. The EXECUTE ANY PROCEDURE system privilege allows calls to any procedure or function, either standalone or packaged, as well as public package variables and cursors, but this is a security risk and should be avoided.

To modify a stored subprogram owned by someone else you need the ALTER ANY PROCEDURE which allows the person to modify ANY procedure owned by ANY user in the database. This is a very wide open privilege that should not be granted in a production database to anyone outside of the DBA group since it could result in loss or damage to application code from either willful or accidental use. It is also a conduit for hackers to access programs and data that should be kept private.

Access to Objects

One of the quirks of creating stored subprograms is that the person developing the subprogram needs to have direct grants to the objects being referenced in the subprogram. Those grants cannot be given through a role.

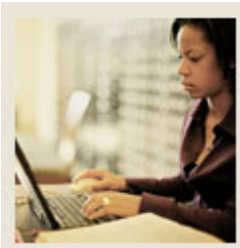
The person calling the subprogram does not need grants to the objects inside the subprograms. The subprogram executes under the privileges of the owner of the object; therefore, subprograms encapsulate security as well – allowing the end user to use the objects references in the subprograms without having direct access themselves.

(Note: There is a way to run subprograms such that the person executing the program must have grants to the objects withing the subprogram. This is called Invoker's Rights but is outside the scope of this course.)

Package Elements

Currently there is no way to grant access to only one component of a package. When the EXECUTE privilege is granted on a package, the grantee has access to all the public elements of the package.

To grant access to only one component of a package, create a procedure wrapper that only calls that one component of the package and then grant EXECUTE on the procedure to the end user.



Section I: Packages

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

a. Create a package called 'Account' containing both the insert_acct_info procedure and the calc_amt_owed function created in Section H.

b. Overload the account package, so that there are now two functions are called calc_amt_owed. The second function should accept the parameters of pidm and term code, and return the sum of the amount owed for that term code.

c. Use what you have learned to add to the exception handler for the insert_acct_info procedure, so that the message displayed indicates the nature of the error (i.e. incorrect parameters).

d. Execute the procedure and both functions from the package.

Section J: Built-In Packages

Lesson: Overview

Jump to TOC

Introduction

Oracle has provided several built-in packages to enhance the use of PL/SQL. Built-in packages can allow you to

- send messages between sessions connected to the same database
- provide PL/SQL equivalents for some DDL statements
- allow scheduling of PL/SQL procedures
- provide screen output in SQL*Plus or SQL Developer
- manipulate large objects

Objectives

This section will examine the following:

- Built-in packages provided by Oracle that enhance PL/SQL

Section contents

Section J: Built-In Packages

Lesson: Oracle Built-In Packages

Jump to TOC

Built-in packages

Now that you can realize how to take advantage of packages in your own development environments, consider the following. Oracle thought that packages were pretty handy as well. In fact, some of the reserved words that you use in the Oracle SQL environment such as DESC and DICT are elements of a package called STANDARD. After several suggestions from the legions of Oracle developers out there, Oracle put together a special set of built-in packages to accommodate for some shortcomings and prevent users from having to build common functionality from scratch.

We will examine a few of these built-in packages in this section. Other built-ins are examined in the remaining sections of this course.

Partial list

The following is a partial list of built-in packages provided by Oracle. Most of these are installed by default when you create a database instance, but in some cases you may have to grant execute privileges to make them available to the user community.

For a complete list of these packages, refer to the Oracle documentation.

Package Name	Description
dbms_alert	Provides support for notification of database events on an asynchronous basis. Registers a process with an alert and then waits for a signal from that alert.
dbms_ crypto	Lets you encrypt and decrypt stored data, can be used in conjunction with PL/SQL programs running network communications, and supports encryption and hashing algorithms.
dbms_ddl	Provides a programmatic access to some of the SQL DDL statements.
dbms_ debug	Implements server-side debuggers and provides a way to debug server-side PL/SQL program units.
dbms_fga	Provides fine-grained access security functions.
dbms_job	Used to submit and manage regularly scheduled jobs for execution inside the database.
dbms_lob	Provides general purpose routines for operations on Oracle Large Object (LOBs) datatypes - BLOB, CLOB (read/write), and BFILEs (read-only).
dbms_ logminer	Provides functions to mine archive logs.
dbms_ metadata	Routines to extract database object definitions from the data dictionary.
dbms_ output	Displays output from PL/SQL programs to the terminal. The “lowest common denominator” debugger mechanism for PL/SQL code.
dbms_pipe	Allows communication between different Oracle sessions through a pipe in the RDBMS shared memory. One of the few ways to share memory-resident data between Oracle sessions.
dbms_ scheduler	Collection of scheduling functions for running scheduled processes.
utl_file	Allows PL/SQL programs to read from and write to operating system files.

Section J: Built-In Packages

Lesson: DBMS_LOB

Jump to TOC

Description

Provides a way to use read and manipulate large objects (BLOB, CLOB) as well as read only access to BFILEs. A selection of the operations in the package is presented below.

OPEN

```
DBMS_LOB.OPEN (lob_loc IN OUT NOCOPY BLOB,
               open_mode IN BINARY_INTEGER);

DBMS_LOB.OPEN (lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
               open_mode IN BINARY_INTEGER);

DBMS_LOB.OPEN (file_loc IN OUT NOCOPY BFILE,
               open_mode IN BINARY_INTEGER := file_readonly);
```

Parameters

Parameter	Value
Loc_loc/File_loc	Name or location of LOB
Open_mode	Mode in which to open LOB: LOB_READONLY or LOB_READWRITE for CLOB/BLOB FILE_READONLY for BFILE

Notes

- It is not mandatory but recommended that LOBs are explicitly opened and closed

ISOPEN

Check to see whether a LOB is open before closing.

```
DBMS_LOB.ISOPEN (lob_loc IN BLOB) RETURN INTEGER;

DBMS_LOB.ISOPEN (lob_loc IN CLOB CHARACTER SET ANY_CS)
RETURN INTEGER;

DBMS_LOB.ISOPEN (file_loc IN BFILE) RETURN INTEGER;
```

Parameters

Parameter	Value
Loc_loc/file_loc	Name or location of LOB

CLOSE

```
DBMS_LOB.CLOSE (lob_loc IN OUT NOCOPY BLOB);

DBMS_LOB.CLOSE (lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS);

DBMS_LOB.CLOSE (file_loc IN OUT NOCOPY BFILE);
```

Parameters

Parameter	Value
Loc_loc/file_loc	Name or location of LOB

Notes

- All open LOBs for the transaction must be closed before issuing a commit or rollback
- If you commit before closing a LOB the LOB is updated but associated indexes are not rendering them invalid.
- When the LOB is closed, the associated indexes are updated.

WRITE

This procedure writes data to an internal LOB from a buffer. This procedure will overwrite any existing data in the LOB. To add to an existing LOB, use the APPEND procedure.

```
DBMS_LOB.WRITE (lob_loc IN OUT NOCOPY BLOB,
                amount IN BINARY_INTEGER,
                offset IN INTEGER,
                buffer IN RAW);

DBMS_LOB.WRITE (lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
                amount IN BINARY_INTEGER,
                offset IN INTEGER,
                buffer IN VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

Parameter	Value
Loc_loc	Name or location of LOB
Amount	Number of bytes or characters to be written
Offset	Starting point in bytes or characters (Default=1)
Buffer	Data to be stored

APPEND

```
DBMS_LOB.APPEND (dest_lob IN OUT NOCOPY BLOB,
                 src_lob IN BLOB);

DBMS_LOB.APPEND (dest_lob IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
                 src_lob IN CLOB CHARACTER SET dest_lob%CHARSET);
```

Parameters

Parameter	Value
Dest_lob	Locator of the interal LOB where the data will be appended.
Src_lob	Locator of the interal LOB serving as the source to be appended.

Notes

Applies to internal LOBs (column in a table) only, and not to external or BFILE LOBs.

You are not required to explicitly OPEN and CLOSE the LOB before appending. However, for performance reasons, it is recommended if you will be doing multiple operations on a LOB.

COPY

Copy all or part of the contents of one internal LOB to another internal LOB.

```
DBMS_LOB.COPY (dest_lob IN OUT NOCOPY BLOB,
               src_lob IN BLOB,
               amount IN INTEGER,
               dest_offset IN INTEGER := 1,
               src_offset IN INTEGER := 1);

DBMS_LOB.COPY (dest_lob IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
               src_lob IN CLOB CHARACTER SET dest_lob%CHARSET,
               amount IN INTEGER,
               dest_offset IN INTEGER := 1,
               src_offset IN INTEGER := 1);
```

Parameter	Value
Dest_lob	Locator of the interal LOB where the data will be copied.
Src_lob	Locator of the interal LOB serving as the source to be copied.
Amount	Number of bytes or characters to copy
Dest_offset	Offset location to put the copy
Src_offset	Offset location to get the copy

GETLENGTH

Obtain the length in bytes or characters of a LOB

```
DBMS_LOB.GETLENGTH (lob_loc IN BLOB) RETURN INTEGER;

DBMS_LOB.GETLENGTH (lob_loc IN CLOB CHARACTER SET ANY_CS)
RETURN INTEGER;

DBMS_LOB.GETLENGTH (file_loc IN BFILE) RETURN INTEGER;
```

ERASE

Clear all or part of an internal LOB.

```
DBMS_LOB.ERASE (lob_loc IN OUT NOCOPY BLOB,
                amount IN OUT NOCOPY INTEGER,
                offset IN INTEGER := 1);

DBMS_LOB.ERASE (lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
                amount IN OUT NOCOPY INTEGER,
                offset IN INTEGER := 1);
```

Parameter	Value
Lob_loc	Locator of the interal LOB.
Amount	Number of bytes or characters to erase
Offset	Starting location to Erase

Notes

Erase does not reclaim the space used by the data that was erased. To reclaim the space use the TRIM operator.

TRIM

```
DBMS_LOB.TRIM (lob_loc IN OUT NOCOPY BLOB,
               newlen IN INTEGER);

DBMS_LOB.TRIM (lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
               newlen IN INTEGER);
```

Parameter	Value
Lob_loc	Locator of the interal LOB.
Newlen	New length in bytes or characters

INSTR

Returns the position of the *nth* occurrence of the pattern starting at the specified offset.

```
DBMS_LOB.INSTR (lob_loc IN BLOB,
                pattern IN RAW,
                offset IN INTEGER := 1,
                nth IN INTEGER := 1) RETURN INTEGER;

DBMS_LOB.INSTR (lob_loc IN CLOB CHARACTER SET ANY_CS,
                pattern IN VARCHAR2 CHARACTER SET lob_loc%CHARSET,
                offset IN INTEGER := 1,
                nth IN INTEGER := 1) RETURN INTEGER;

DBMS_LOB.INSTR (file_loc IN BFILE,
                pattern IN RAW,
                offset IN INTEGER := 1,
                nth IN INTEGER := 1) RETURN INTEGER;
```

SUBSTR

Return the amount of bytes or characters from the LOB starting at the offset.

```
DBMS_LOB.SUBSTR (lob_loc IN BLOB,
                 amount IN INTEGER := 32767,
                 offset IN INTEGER := 1) RETURN RAW;

DBMS_LOB.SUBSTR (lob_loc IN CLOB CHARACTER SET ANY_CS,
                 amount IN INTEGER := 32767,
                 offset IN INTEGER := 1)
                 RETURN VARCHAR2 CHARACTER SET lob_loc%CHARSET;

DBMS_LOB.SUBSTR (file_loc IN BFILE,
                 amount IN INTEGER := 32767,
                 offset IN INTEGER := 1) RETURN RAW;
```

Section J: Built-In Packages

Lesson: DBMS_LOB (Continued)

Jump to TOC

Initializing LOBs

When inserting rows into a table with a LOB value, the LOB column must be initialized You can initialize a BLOB column value by using the built-in function EMPTY_BLOB(). A CLOB or NCLOB column value can be initialized by using the built-in function EMPTY_CLOB().

EMPTY_BLOB();
EMPTY_CLOB():

DBMS_LOB Examples

create table my_lob_table (lob_row_number number(3),
lob_column CLOB);

```
SQL> INSERT into my_lob_table values (3,empty_clob());

DECLARE
   lv_lob_holder    CLOB;
   lv_lob_size      NUMBER;
   lv_lob_buffer    VARCHAR2(4000);
BEGIN
   lv_lob_buffer := 'To Be Or Not To Be, That Is The Question';
   lv_lob_size := length(lv_lob_buffer);
   DBMS_OUTPUT.put_line('Buffer size: ' || lv_lob_size);

   SELECT lob_column
     INTO lv_lob_holder
     FROM my_lob_table
    WHERE lob_row_number = 3 FOR UPDATE;

   DBMS_LOB.WRITE(lv_lob_holder,lv_lob_size,1,lv_lob_buffer);
   lv_lob_size := DBMS_LOB.GETLENGTH(lv_lob_holder);

   DBMS_OUTPUT.put_line('LOB size: ' || lv_lob_size);
   COMMIT;
END;
/
```

Section J: Built-In Packages

Lesson: DBMS_RANDOM

Jump to TOC

Description

This package can calculate either random numbers in the range -2^31 to 2^31 or between 0 and 1 with 38 digits of precision. It can also be used to generate random string values. This should not be used to generate encryption keys – use DBMS_CRYPTO for security encryption.

NORMAL

Return a number in the normal distribution (-2^31 to 2^31).

```
   DBMS_RANDOM.NORMAL RETURN NUMBER;

SQL> select dbms_random.normal from dual;

    NORMAL
----------
1.07677152
```

SEED

Setup the SEED value from which the random number will be generated. You can generate the same 'random' numbers for testing by using the same seed values in DBMS_RANDOM.SEED. If you don't 'SEED' the generator, Oracle will supply a default seed value.

```
DBMS_RANDOM.SEED (seed IN BINARY_INTEGER);
DBMS_RANDOM.SEED (seed IN VARCHAR2);
```

Characters can be up to 2000 in length.

STRING

Returns a random character string up to 4000 characters in length.

```
DBMS_RANDOM.STRING (opt IN CHAR,
                    len IN NUMBER) RETURN VARCHAR2;
```

Parameters

Parameter	Value
Opt	Specifies what the returning string looks like: ■ 'u' or 'U' – returns uppercase alpha characters ■ 'l' or 'L' – returns lowercase alpha characters ■ 'a' or 'A' – returns mixed case alpha characters ■ 'x' or 'X' – returns uppercase alpha-numeric ■ 'p' or 'P' – returns any printable characters. Default is uppercase alpha characters.
Len	Length of the return string

Examples

```
SQL> select dbms_random.string('X',10) from dual;

DBMS_RANDOM.STRING('X',10)
--------------------------------------------------------------------
47C2FZNLF3

SQL> select dbms_random.string('P',10) from dual;

DBMS_RANDOM.STRING('P',10)
--------------------------------------------------------------------
R*:2&:XOc>
```

VALUE

Generates a number between 0 and 1 with 38 digits of precision or between a low and high specified value. Note: random number can be either the low or high value.

```
DBMS_RANDOM.VALUE  RETURN NUMBER;

DBMS_RANDOM.VALUE(low IN NUMBER,
                  high IN NUMBER) RETURN NUMBER;

SQL> select dbms_random.value(4,4000) from dual;

DBMS_RANDOM.VALUE(4,4000)
-------------------------
               3920.69794

SQL> select dbms_random.value from dual;

DBMS_RANDOM.VALUE
----------------------------------------
0.944907158520214139338722542107819248 39
```

Section J: Built-In Packages

Lesson: DBMS_OUTPUT

Jump to TOC

Description

The built-in package DBMS_OUTPUT is used to send output to the screen (or a file when used in a script). For the package to be enabled, you first must enter the following at the SQL prompt:

```
SET SERVEROUTPUT ON
```

You can also put this into your profile (login.sql) so that SERVEROUTPUT is on whenever you enter SQL*Plus.

The same can be accomplished when using SQL Developer by clicking on the DBMS Output tab, then on the Enable button: 

Within a program, you must initially enable DBMS_OUTPUT using the ENABLE procedure:

```
DBMS_OUTPUT.ENABLE;
```

Available procedures

The following procedures are available to you for output:

Function/Procedure	Description
ENABLE	Enable message output
DISABLE	Disable message output
PUT_LINE	Place a line in the buffer
PUT	Place a partial line in the buffer
NEW_LINE	Terminate a line created with PUT
GET_LINE	Retrieve one line of information from buffer
GET_LINES	Retrieve array of lines from buffer

Example

```
/*Handles the invalid session error differently from all other
errors.  Is not already predefined, so must be declared. */
DECLARE
   invalid_session EXCEPTION;
   PRAGMA          EXCEPTION_INIT (invalid_session, -22);
   lv_sqlcode      NUMBER;
   lv_sqlerrm      CHAR(55);
BEGIN
   DBMS_OUTPUT.ENABLE;
   INSERT INTO swriden (swriden_pidm, swriden_id, swriden_last_name,
                        swriden_activity_date)
          VALUES (1234, '56', 'Peterson',SYSDATE);
EXCEPTION
  WHEN invalid_session THEN
      DBMS_OUTPUT.PUT_LINE('Invalid session ID. Access
 Denied. Contact Information Services at 123-4567');
  WHEN OTHERS THEN
      lv_sqlcode := SQLCODE;
      lv_sqlerrm := SUBSTR(SQLERRM, 1, 55);
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE(lv_sqlcode||' '||lv_sqlerrm);
END;
/
```

Note: You can also use the DBMS_OUTPUT package for basic reports, though the package was not created with this intention.

Section J: Built-In Packages

Lesson: DBMS_SESSION

Jump to TOC

Description

This package has many useful functions. The ones presented here can be used for debugging as well as security.

SET_ROLE

Calling DBMS_SESSION.SET_ROLE is equivilant to issuing the command 'alter session set role…;'. It can be used in applications like Oracle Forms to enable a role for a particular form that may differ from other forms. Banner uses this command to invoke the role assigned to a user through GSASECR when each Oracle Form is opened.

```
DBMS_SESSION.SET_ROLE (role_cmd VARCHAR2);
```

The role_cmd parameter is the name of the role being invoked. It may also include the password to invoke a password protected role.

SET_CONTEXT

The procedure sets context or application-defined attributes about a session that can be used in programs and processes to take action based on that information. For example, in Banner it is used in Fined Grained Access to set attributes about a session which determine what security that session is allowed on certain objects. Once these values have been set for a session they can be accessed via another built-in called SYS_CONTEXT and used to take certain actions as well as deny or allow access to data.

```
DBMS_SESSION.SET_CONTEXT (namespace VARCHAR2,
                          attribute VARCHAR2,
                          value VARCHAR2,
                          username VARCHAR2,
                          client_id VARCHAR2 );
```

Parameter	Description
Namespace	Name of a related group of attributes (maxsize 30)
Attribute	Name of the parameter being set (maxsize 30)
Value	Value of the parameter or attribute (maxsize 4k)
Username	For a Global Namespace, compared to the current database user. Default = NULL
Client_id	For a Global Namespace, compared to the current database user's internal ID. Default = NULL

Example (from package GOKFGAC)

```
FUNCTION f_query_predicate (
     p_objectname VARCHAR2,
     p_crud       VARCHAR2,
     p_fgac_user  VARCHAR2)
  RETURN VARCHAR2 IS

 l_predicate    VARCHAR2(4000);
 l_schema       VARCHAR2(30) := '';
 d_ctx          VARCHAR2(4000);
 lv_pii_cnt     NUMBER := 0;
 begin

 -- clear the saved predicate so the same result isnt viewed over and over
  CASE p_crud
      WHEN  D_INS then
        d_ctx := 'ctx_fg_' || LOWER(p_objectname) || '_ins';
      WHEN  D_UPD then
        d_ctx := 'ctx_fg_' || LOWER(p_objectname) || '_upd';
      WHEN  D_DEL then
        d_ctx := 'ctx_fg_' || LOWER(p_objectname) || '_del';
      WHEN  D_SEL then
        d_ctx := 'ctx_fg_' || LOWER(p_objectname) || '_sel';
      ELSE
        d_ctx := 'ctx_fg_' || LOWER(p_objectname) || '_sel';
  END CASE;
 dbms_session.set_context('g$_vbsi_context',d_ctx, ''  );

 query_user := p_fgac_user;

 l_predicate  := gokfgac.f_common_predicate (l_schema ,
                                    p_objectname,
                                    p_crud ,
                                    p_fgac_user);

 -- reset if its a pii table
 SELECT NVL(COUNT(*),0) INTO lv_pii_cnt FROM gorfdpi
 WHERE gorfdpi_table_name = p_objectname;
 IF  lv_pii_cnt > 0 THEN
   gokfgac.p_get_user_info(gb_common.f_sct_user) ;
 END IF;
 -- reset saved predicate
 dbms_session.set_context('g$_vbsi_context',d_ctx, '' );

 IF l_predicate = NULL_PRED THEN
   RETURN (NULL);
 ELSE
   RETURN l_predicate;
 END IF;
END f_query_predicate;
```

Other CONTEXT Related

Other useful processes inside the package controlling the setting or removing of CONTEXT information include:

```
DBMS_SESSION.CLEAR_ALL_CONTEXT (namespace VARCHAR2);

DBMS_SESSION.CLEAR_CONTEXT (namespace VARCHAR2,
                            client_identifier VARCHAR2
                            attribute VARCHAR2);
```

```
For listing the values of a Context:
TYPE AppCtxRecTyp IS RECORD (namespace VARCHAR2(30),
                             attribute VARCHAR2(30),
                             value VARCHAR2(256));

TYPE AppCtxTabTyp IS TABLE OF AppCtxRecTyp INDEX BY BINARY_INTEGER;

DBMS_SESSION.LIST_CONTEXT (list OUT AppCtxTabTyp,
                           size OUT NUMBER);
```

SET_SQL_TRACE

This is equivalent to issuing the command 'alter session set sql_trace = true/false;'. It creates a trace file in the database UDUMP directory for the session (the OS Process ID is included in the name of the trace file) and writes tracing information on all SQL executed between the time trace is turned on until it is turned off. It can be used for debugging purposes to trace sections of code.

```
DBMS_SESSION.SET_SQL_TRACE (sql_trace boolean);
```

Parameter	Description
Sql_trace	TRUE to turn on; FALSE to turn off

Section J: Built-In Packages

Lesson: SYS_CONTEXT

Jump to TOC

Description

SYS_CONTEXT is a built-in function that returns information about the value of an attribute in a namespace. It can be used to obtain information on a default namespace called USERENV or from an application defined namespace.

Example

```
CONNECT HRISUSR/U_PICK_IT
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
----------------------------------------------------
HRISUSR
```

Common Attributes for USERENV

(Refer to SQL Reference Manual for a complete list)

Attribute	Description
Action	Current action session is taking (e.g. SELECT, INSERT, UPDATE, EXECUTE)
DB_Domain	Domain of database
DB_Name	Name of database
Host	Host name of the client's machine
Instance_name	Name of instance connected (RAC)
IP_Address	IP address of the client's machine
Language	Language settings for session (territory.characterset)
NLS_Date_Format	Date format for the session
OS_User	Name user authenticated to their machine
Session_User	Name user authenticated to the database
SID	Session ID in database for user

```
SQL> select sys_context('userenv','language') from dual;

SYS_CONTEXT('USERENV','LANGUAGE')
----------------------------------------------------------------
AMERICAN_AMERICA.WE8MSWIN1252
```

Section J: Built-In Packages

Lesson: DBMS_SCHEDULER

Jump to TOC

Description (10g+)

This package allows you to schedule processes to run at certain intervals or based on events. A selection of the procedures in the package is show here. Jobs can even be chained together to provide a dependent set of processes. For a full list consult the PL/SQL Reference Manual.

This is replacing DBMS_JOB which is still available for backward compatibility.

CREATE_PROGRAM

This procedure creates a process that will run under the scheduler. It will not run until it is associated with a schedule using the CREATE_JOB procedure.

```
DBMS_SCHEDULER.CREATE_PROGRAM (program_name IN VARCHAR2,
                  program_type IN VARCHAR2,
                  program_action IN VARCHAR2,
                  number_of_arguments IN PLS_INTEGER DEFAULT 0,
                  enabled IN BOOLEAN DEFAULT FALSE,
                  comments IN VARCHAR2 DEFAULT NULL);
```

Parameter	Description
Program_name	User supplied name; cannot be same as another object in user's schema
Program_type	Options: • PLSQL_BLOCK – anonymous block you specify • STORED_PROCEDURE – PL/SQL or Java stored procedure or C program; Functions and procedures with OUT or INOUT not allowed • EXECUTABLE – external program available from O/S command line
Program_action	For PLSQL_BLOCK – the actual anonymous block including BEGIN and END, ending in a semicolon (;) For STORED_PROCEDURE – name of the stored procedure including schema if applicable For EXECUTABLE – executable name including full path and any arguments
Number_of_arguments	Default = 0, max = 255; ignored for PLSQL_BLOCK
Enabled	Default=False; can be enabled with ENABLE procedure; must be enabled to run
Comments	User supplied description of program

Permissions

- The user must have been granted CREATE JOB privilege to run this procedure.
- For another user to execute your job they must have EXECUTE privilege on the job.

Example

```
SQL> BEGIN
   DBMS_SCHEDULER.CREATE_PROGRAM (
      program_name => 'plsql_block_program',
      program_type => 'PLSQL_BLOCK',
      program_action =>
        'BEGIN
         INSERT INTO TEMP (col1, col3, message)
            (SELECT COUNT(*), SYSDATE,
            ''Swriden record count'' FROM TRAIN01.SWRIDEN);
         END;',
      enabled => TRUE,
      comments => 'Program to track growth of swriden table');
END;
/
```

CREATE_SCHEDULE

This procedure creates a job schedule that is independent of any single job. It can be used for multiple jobs and provides and easy mechanism to change schedules for multiple jobs at one time for all jobs assigned the schedule.

```
DBMS_SCHEDULER.CREATE_SCHEDULE (
             schedule_name IN VARCHAR2,
             start_date IN TIMESTAMP WITH TIMEZONE DEFAULT NULL,
             repeat_interval IN VARCHAR2,
             end_date IN TIMESTAMP WITH TIMEZONE DEFAULT NULL,
             comments IN VARCHAR2 DEFAULT NULL);
```

Parameter	Description
Schedule_name	User supplied name; cannot be same as another object in user's schema
Start_date	Date to start schedule; Can be omitted for repeating schedules and is, therefore, derived from the interval
Repeat_interval	How often schedule should repeat **
End_date	Date schedule will stop running; blank = forever
Comments	User supplied description of the schedule

** See below for Interval Syntax

Privileges

This procedure requires the CREATE JOB privilege to run. Schedules are created with PUBLIC privileges and therefore, can be used by anyone once created.

Schedule Interval Syntax (partial listing)

```
regular_schedule = frequency_clause
[";" interval_clause] [";" bymonth_clause] [";" byweekno_clause]
[";" byyearday_clause] [";" bydate_clause] [";" bymonthday_clause]
[";" byday_clause] [";" byhour_clause] [";" byminute_clause]
[";" bysecond_clause]

frequency_clause = "FREQ" "=" ( predefined_frequency | user_defined_frequency )
       predefined_frequency = "YEARLY" | "MONTHLY" | "WEEKLY" | "DAILY" |
       "HOURLY" | "MINUTELY" | "SECONDLY"

interval_clause = "INTERVAL" "=" [1 through 999]
bymonth_clause = "BYMONTH" "=" monthlist
       monthlist = monthday ( "," monthday)*
       month = numeric_month | char_month
       numeric_month = 1 | 2 | 3 ... 12
       char_month = "JAN" | "FEB" | "MAR" | "APR" | "MAY" | "JUN" |"JUL" | "AUG" |
       "SEP" | "OCT" | "NOV" | "DEC"
byweekno_clause = "BYWEEKNO" "=" weeknumber_list
       weeknumber_list = weeknumber ( "," weeknumber)*
       weeknumber = [minus] weekno
       weekno = 1 through 53
byyearday_clause = "BYYEARDAY" "=" yearday_list
       yearday_list = yearday ( "," yearday)*
       yearday = [minus] yeardaynum
       yeardaynum = 1 through 366
bydate_clause = "BYDATE" "=" date_list
       date_list = date ( "," date)*
       date = [YYYY]MMDD
bymonthday_clause = "BYMONTHDAY" "=" monthday_list
       monthday_list = monthday ( "," monthday)*
       monthday = [minus] monthdaynum
       monthdaynum = 1 through 31
byday_clause = "BYDAY" "=" byday_list
       byday_list = byday ( "," byday)*
       byday = [weekdaynum] day
       weekdaynum = [minus] daynum
       daynum = 1 through 53 /* if frequency is yearly */
       daynum = 1 through 5 /* if frequency is monthly */
       day = "MON" | "TUE" | "WED" | "THU" | "FRI" | "SAT" | "SUN"
byhour_clause = "BYHOUR" "=" hour_list
       hour_list = hour ( "," hour)*
       hour = 0 through 23
byminute_clause = "BYMINUTE" "=" minute_list
       minute_list = minute ( "," minute)*
       minute = 0 through 59
bysecond_clause = "BYSECOND" "=" second_list
       second_list = second ( "," second)*
       second = 0 through 59

In calendaring syntax, * means 0 or more.
```

Schedule Interval Syntax (Continued)

Bold items from above are explained in more detail in cases where their definition may not be obvious.

Parameter	Description
INTERVAL	How often recurrence repeats. (e.g. 1 = every second for secondly, every day for daily, etc.)
BYWEEKNO	Only valid for YEARLY. Week of the year following the ISO-8601 standard where a week starts on Monday and ends on Sunday. Part of week 1 may be in a previous year and part of week 52/53 might be in a following year. Negative numbers are valid "[minus] weekno". Indicates weeks back from the end of the year.
BYYEARDAY	Day of the year. Takes into account leap years. Can be negative – days back from the end of the year.
BYMONTHDAY	Day of the month. Can be negative – days back from the last day of the month (e.g. -1 = last day of the month; -3 = third to the last day of the month).

Examples

```
DBMS_SCHEDULER.create_schedule (
             schedule_name => 'hourly_schedule',
             start_date => SYSTIMESTAMP,
             repeat_interval => 'freq=hourly; byminute=0',
             end_date => NULL,
             comments => 'Repeats hourly, on the hour, forever.');

DBMS_SCHEDULER.create_schedule (
             schedule_name => 'monthly_schedule',
             start_date => SYSTIMESTAMP,
             repeat_interval => 'freq=monthly; bymonthday=-1',
             end_date => NULL,
             comments => 'Repeats monthly, on the last day of the
month, forever.');
```

CREATE_JOB

This is an overloaded procedure which can be called to create a wholly independent job or to combine predefined programs and schedules into a running job. Jobs can be based on a schedule or on an event. Refer to the PL/SQL Reference Manual for information on creating jobs based on events.

Syntax

Creates a job in a single call without using an existing program or schedule:

```
DBMS_SCHEDULER.CREATE_JOB (
                        job_name IN VARCHAR2,
                        job_type IN VARCHAR2,
                        job_action IN VARCHAR2,
                        number_of_arguments IN PLS_INTEGER DEFAULT 0,
                        start_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
                        repeat_interval IN VARCHAR2 DEFAULT NULL,
                        end_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
                        job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
                        enabled IN BOOLEAN DEFAULT FALSE,
                        auto_drop IN BOOLEAN DEFAULT TRUE,
                        comments IN VARCHAR2 DEFAULT NULL);
```

Creates a job using a named schedule object and a named program object:

```
DBMS_SCHEDULER.CREATE_JOB (
                        job_name IN VARCHAR2,
                        program_name IN VARCHAR2,
                        schedule_name IN VARCHAR2,
                        job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
                        enabled IN BOOLEAN DEFAULT FALSE,
                        auto_drop IN BOOLEAN DEFAULT TRUE,
                        comments IN VARCHAR2 DEFAULT NULL);
```

Creates a job using a named program object and an inlined schedule:

```
DBMS_SCHEDULER.CREATE_JOB (
                        job_name IN VARCHAR2,
                        program_name IN VARCHAR2,
                        start_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
                        repeat_interval IN VARCHAR2 DEFAULT NULL,
                        end_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
                        job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
                        enabled IN BOOLEAN DEFAULT FALSE,
                        auto_drop IN BOOLEAN DEFAULT TRUE,
                        comments IN VARCHAR2 DEFAULT NULL);
```

Section J: Built-In Packages

Lesson: DBMS_SCHEDULER (continued)

Jump to TOC

Creates a job using a named schedule object and an inlined program:

```
DBMS_SCHEDULER.CREATE_JOB (
                        job_name IN VARCHAR2,
                        schedule_name IN VARCHAR2,
                        job_type IN VARCHAR2,
                        job_action IN VARCHAR2,
                        number_of_arguments IN PLS_INTEGER DEFAULT 0,
                        job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
                        enabled IN BOOLEAN DEFAULT FALSE,
                        auto_drop IN BOOLEAN DEFAULT TRUE,
                        comments IN VARCHAR2 DEFAULT NULL);
```

Parameter	Description
Job_name	User supplied name of the job. Used to stop/start job.
Job_type	• PLSQL_BLOCK – anonymous block you specify • STORED_PROCEDURE – PL/SQL or Java stored procedure or C program; Functions and procedures with OUT or INOUT not allowed • EXECUTABLE – external program available from O/S command line • CHAIN – part of a chain of jobs
Job_action	For PLSQL_BLOCK – the actual anonymous block including BEGIN and END, ending in a semicolon (;) For STORED_PROCEDURE – name of the stored procedure including schema if applicable For EXECUTABLE – executable name including full path and any arguments For CHAIN – name of a chain object
Num_of_arguments	Number of arguments passed to job (between 0 and 255). Not available for job_type CHAIN.
Program_name	Name of program created with CREATE_PROGRAM
Start_date	Date to start schedule; Can be omitted for repeating schedules and is, therefore, derived from the interval
Repeat_interval	How often schedule should repeat (see above for syntax)
Schedule_name	Name of schedule created with CREATE_SCHEDULE
End_date	Date job will stop running. The job becomes disabled.
Comments	User supplied description of job
Enabled	Default=False; can be enabled with ENABLE procedure; must be enabled to run
Auto_drop	Should the job be dropped when end_date or last iteration reached? Default=False.

Permissions

The user must have the CREATE JOB system privilege to execute this procedure. In addition, if the job_type is EXTERNAL the user must have CREATE EXTERNAL JOB permission for the job to be enabled and/or run.

Examples

```
DBMS_SCHEDULER.create_job (
   job_name => 'count_swriden_job',
   program_name => 'plsql_block_program',
   schedule_name => 'monthly_schedule',
   enabled => TRUE,
   comments => 'Job using existing program and schedule.');

DBMS_SCHEDULER.create_job (
   job_name => 'test_program_schedule_job',
   job_type => 'stored_procedure',
   job_action => 'sp_my_procedure',
   number_of_arguments => 1,
   schedule_name => 'hourly_schedule',
   enabled => FALSE,
   comments => 'Job using inline program and schedule.');
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE (
   job_name => 'test_program_schedule_job',
   argument_position => 1,
   argument_value => 'SYSDATE');
DBMS_SCHEDULER.ENABLE (
   name => 'test_program_schedule_job');

DBMS_SCHEDULER.create_job (
   job_name => 'self_contained_job',
   job_type => 'EXECUTABLE',
   job_action => '/app/oracle/scripts/monthly_job.sh',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'freq=monthly; bymonthday=15',
   end_date => NULL,
   enabled => TRUE,
   comments => 'Job created with inline code and schedule');
```

Other Procedures

Procedures in the DBMS_SCHEDULER package that start, stop, and drop scheduled items.

Disable

```
DBMS_SCHEDULER.DISABLE (name IN VARCHAR2,
                        force IN BOOLEAN DEFAULT FALSE);
```

If force is set to FALSE and the job is currently running, an error is returned.
If force is set to TRUE, the job is disabled, but the currently running instance is allowed to finish.

Enable

```
DBMS_SCHEDULER.ENABLE (name IN VARCHAR2);
```

Name can be a comma separated list of names.

Drop Job

```
DBMS_SCHEDULER.DROP_JOB (job_name IN VARCHAR2,
                         force IN BOOLEAN DEFAULT FALSE);
```

If force is set to FALSE, and the job is running, the call results in an error.
If force is set to TRUE, the Scheduler first attempts to stop the running job (by issuing the STOP_JOB call with the force flag set to false), and then drops the job.

Drop Program

```
DBMS_SCHEDULER.DROP_PROGRAM (program_name IN VARCHAR2,
                             force IN BOOLEAN DEFAULT FALSE);
```

If force is set to FALSE, the program cannot be referenced by any other job, otherwise an error will occur.
If force is set to TRUE, all jobs referencing the program are disabled before dropping the program.
Running jobs that point to the program are not affected by the DROP_PROGRAM call, and are allowed to continue.

Drop Schedule

```
DBMS_SCHEDULER.DROP_SCHEDULE (schedule_name IN VARCHAR2,
                              force IN BOOLEAN DEFAULT FALSE);
```

If force is set to FALSE, the schedule cannot be referenced by any other job, otherwise an error will occur.
If force is set to TRUE, any jobs that use this schedule will be disabled before the schedule is dropped
Running jobs and open windows that point to the schedule are not affected.

Run Job

Use to run a job immediately. If use_current_session is FALSE and scheduled job is running run_job will fail.

```
DBMS_SCHEDULER.RUN_JOB (job_name IN VARCHAR2,
                        use_current_session IN BOOLEAN DEFAULT
                        TRUE);
```

Stop Job

Stop a running job. Using the 'force' option requires the system privilege MANAGE SCHEDULER.

```
DBMS_SCHEDULER.STOP_JOB (job_name IN VARCHAR2
                         force IN BOOLEAN DEFAULT FALSE);
```

Dictionary Tables

USER_SCHEDULER_JOBS
USER_SCHEDULER_JOB_ARGS
USER_SCHEDULER_JOB_LOG
USER_SCHEDULER_JOB_RUN_DETAILS
USER_SCHEDULER_RUNNING_JOBS

Section J: Built-In Packages

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

Write a script that generates a random number and a random string. Use the package DBMS_RANDOM to generate the values and DBMS_OUTPUT to display them.

Exercise 2

Write a script to use the SYS_CONTEXT built in and at least 3 of the attributes to display information about your session. Display those values using DBMS_OUTPUT.

Exercise 3

Create a table with a CLOB column and an Identifier column. Insert a row into the table using the EMPTY_CLOB() built in.

Write a string to the CLOB record you just created. Append the same value to that CLOB. Show the length of the LOB after the write and append.

Run your procedure again. What happens to the length and why?

Section K: Database Triggers

Lesson: Overview

Jump to TOC

Introduction

Database triggers are similar to stored procedures and functions in the fact that they are written with PL/SQL code, and are stored within the database. However, database triggers are "fired" due to a database event, rather than being explicitly called from a PL/SQL block.

Objectives

This section will examine the following:

- The different types of trigger events
- Restrictions on triggers
- View trigger code
- Drop or disable a trigger

Section contents

Section K: Database Triggers

Lesson: Trigger Events

Jump to TOC

Triggers

A trigger can be associated with a Table, View, or Event. We will be working primarily with Table triggers in this section. Table triggers can be fired just before or after inserts, updates, and deletes on a table. In addition, the triggers can fire for each row, or each statement.

Capabilities

Triggers can be used to provide any of the following sample functionality:

- Write historical data to history tables
- Write audit records
- Compute values for columns
- Assign system level values to columns (sysdate, user, data source, sequence values)
- Prevent invalid data from entering database (start_date > end_date)
- Prevent unauthorized data changes
- Restrict table access to business hours
- Update associated tables
- Enforce business rules that cannot be accomplished through referential integrity

Syntax

```
CREATE [OR REPLACE] TRIGGER trigger_name
       { BEFORE | AFTER } triggering_event
       ON table_reference
       [ FOR EACH ROW [ WHEN trigger_condition ]]
       trigger_body;
```

Parameter	Description
Trigger_name	User supplied name of trigger
Triggering_event	Insert, Update, Delete, Instead of (views)
Table_reference	Name of table or view
Trigger_condition	Condition under which trigger fires (e.g. salary > 50000)
Trigger_body	Standard PL/SQL code

Section K: Database Triggers

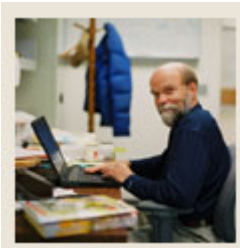
Lesson: Trigger Events (continued)

Jump to TOC

Statement vs Row

A table trigger can fire once per event (statement level) or once per row affected by the event (row level). A statement level trigger would only execute its code once regardless of how many rows were inserted, updated, or deleted. A row level trigger executes its code for each row affected by the insert, update, or delete.

A statement level trigger is created by default. To create a row level tigger, add the FOR EACH ROW clause to the trigger creation syntax.



Section K: Database Triggers

Lesson: Old and New in Row-Level Triggers

Jump to TOC

Handling :OLD and :NEW

A row-level trigger fires once per row processed by the triggering statement. Within the trigger, you can access the old and new values of the row that is currently being processed. This is accomplished by referencing the fields of the pseudo-records :OLD and :NEW. These values are used as bind variables that are populated when the trigger is executed.

The datatype of each pseudo-record is as follows:

```
triggering_table%ROWTYPE;
```

Example using :OLD:

```
CREATE OR REPLACE TRIGGER update_swriden
    AFTER UPDATE ON swriden
    FOR EACH ROW
BEGIN
   INSERT INTO swriden_history
   (swriden_hist_pidm, swriden_hist_id, swriden_hist_first_name,
    swriden_hist_last_name, swriden_hist_change_ind,
    swriden_hist_activity_date)
   VALUES (:OLD.swriden_pidm, :OLD.swriden_id,
           :OLD.swriden_first_name, :OLD.swriden_last_name,
           :OLD.swriden_change_ind, :OLD.swriden_activity_date);
END update_swriden;
/
```

The following statements would cause the above trigger to fire:

```
update swriden set swriden_first_name = 'Joe'
 where swriden_pidm = 12340;
commit;
```

Example using :NEW:

```
CREATE SEQUENCE pidm_sequence START WITH 20000;

CREATE OR REPLACE TRIGGER gen_pidm
   BEFORE INSERT ON swriden
   FOR EACH ROW
BEGIN
    SELECT pidm_sequence.NEXTVAL
     INTO :NEW.swriden_pidm
     FROM dual;
END gen_pidm;
/
```

Section K: Database Triggers

Lesson: Restrictions on Triggers

Jump to TOC

Multiple Triggers per Table

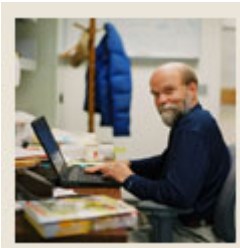
You can have multiple triggers on a table. There could be a separate trigger for Updates, Inserts, and Deletes.

A table can also have one trigger that handles all three options. A trigger like this would look similar to:

```
CREATE OR REPLACE TRIGGER TIUD_SWRIDEN
        BEFORE INSERT OR UPDATE OR DELETE
        ON SWRIDEN
        REFERENCING OLD AS OLD NEW AS NEW
        FOR EACH ROW
BEGIN
  IF INSERTING then
     -- take some actions…..
  ELSIF UPDATING then
     -- take some other actions…..
  ELSIF DELETING then
     -- take the last alternate actions….
  END IF;
END;
/
```

In addition you can have multiples of each type of trigger on a table. You could have two triggers for updates, three for inserts and one for deletes. Why would you want to do that? The application may place triggers on a table. Instead of modifying baseline code, you could *add* another trigger of the same type to incorporate your own code. However, Oracle will not guarantee the order in which multiple triggers fire. Therefore, do not create a trigger that is dependent on the results of another trigger.

Note: The above example uses the REFERENCING clause with the default correlation names OLD and NEW. Alternate correlation names can be specified if desired. (i.e. REFERENCING OLD AS BEFORE NEW AS AFTER)



Section K: Database Triggers

Lesson: Restrictions on Triggers (continued)

Jump to TOC

Restrictions

- Triggers may not issue a transactional statement, such as COMMIT, ROLLBACK, or SAVEPOINT unless they have been defined with the AUTONOMOUS TRANSACTION directive.
- Any procedures that are called from the trigger cannot contain any transactional statements
- The trigger body cannot declare any LONG or LONG RAW variables
- The :NEW and :OLD keywords cannot refer to a LONG or LONG RAW column
- Triggers cannot exceed 32k in size
- DDL Statements are not allowed in a trigger

Section K: Database Triggers

Lesson: Autonomous Transactions

Jump to TOC

Description

The PRAGMA AUTONOMOUS_TRANSACTION directive can be given to a trigger or procedure, package, or anonymous block to indicate that the actions in that PL/SQL act as a separate transaction from the one calling the PL/SQL.

This allows you to perform actions like writing errors to an error table even though the transaction causing the error may roll back any changes. Previously, even if you wrote information to an error table in a trigger, it would be rolled back if the surrounding transaction was rolled back. In addition, without this you had to rely on built-ins like DBMS_OUTPUT to display error information.

Commands

The following commands are allowed in Autonomous Transactions for the currently active block of code only:

- SET TRANSACTION
- COMMIT
- ROLLBACK
- SAVEPOINT
- ROLLBACK TO SAVEPOINT

Note that a rollback in the main transaction will not undo anything committed in the Autonomous Transaction. Therefore, if you are, for example, writing audit records in an Autonomous Transaction and the transaction gets rolled back, the audit record would still exist.

Syntax

```
CREATE OR REPLACE TRIGGER st_track_ssn_changes
   AFTER UPDATE ON swriden
   FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;

BEGIN
   INSERT INTO audit_table
      (audit_id, table_name, old_value, new_value,
       message, activity_date)
   VALUES
      (audit_seq.NEXTVAL, 'SWRIDEN', :OLD.swriden_id, :NEW.swriden_id,
       'SWRIDEN_ID changed', SYSDATE);
   COMMIT;

END;
/
```

Section K: Database Triggers

Lesson: The WHEN Clause

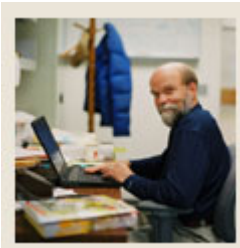
Jump to TOC

WHEN clause

The WHEN clause is valid for row-level triggers only, and is used to restrict when the trigger fires.

```
  CREATE OR REPLACE TRIGGER check_amount
     BEFORE UPDATE OF twraccd.amount ON twraccd
     FOR EACH ROW
     WHEN (NEW.twraccd_amount IS NULL)
  BEGIN
  … trigger body
  END;
/
```

Note: Colons in front of the NEW and OLD pseudo-columns are only used within a trigger body. These keywords or correlation names are not considered bind variables when specified in the WHEN clause, and so are not preceded by a colon (:). Remember to omit the colon in front of the OLD and NEW pseudo-columns within the WHEN clause.



Section K: Database Triggers

Lesson: Viewing Stored Trigger Code

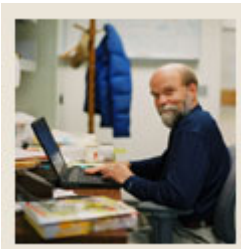
Jump to TOC

USER_ TRIGGERS

Trigger code is not stored in the USER_SOURCE view as it is for procedures, packages, and functions. View the trigger code by querying the database views, USER_TRIGGERS (under your own schema) or ALL_TRIGGERS (any trigger that your schema has access to).

You can view the trigger information as well as the underlying code by querying against the database view, USER_TRIGGERS.

```
SQL> DESC USER_TRIGGERS;

 Name                            Null?    Type
 ------------------------------- -------- ----
 TRIGGER_NAME                             VARCHAR2(30)
 TRIGGER_TYPE                             VARCHAR2(16)
 TRIGGERING_EVENT                         VARCHAR2(227)
 TABLE_OWNER                              VARCHAR2(30)
 BASE_OBJECT_TYPE                         VARCHAR2(16)
 TABLE_NAME                               VARCHAR2(30)
 COLUMN_NAME                              VARCHAR2(4000)
 REFERENCING_NAMES                        VARCHAR2(128)
 WHEN_CLAUSE                              VARCHAR2(4000)
 STATUS                                   VARCHAR2(8)
 DESCRIPTION                              VARCHAR2(4000)
 ACTION_TYPE                              VARCHAR2(11)
 TRIGGER_BODY                             LONG

SELECT trigger_type, table_name, triggering_event
  FROM user_triggers
 WHERE trigger_name = 'UPDATE_SWRIDEN';
```

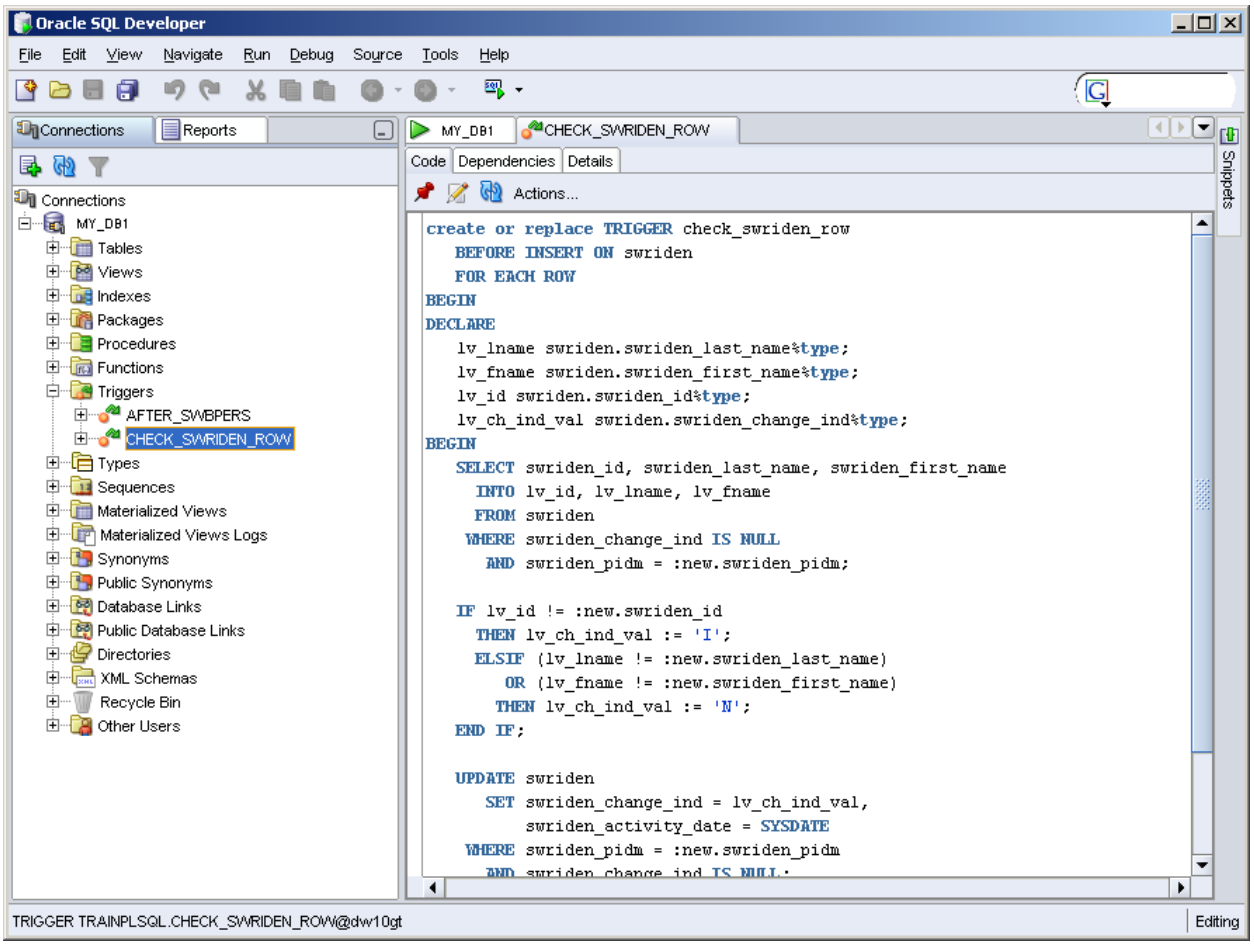


Section K: Database Triggers

Lesson: View Trigger Code (continued)

Jump to TOC

Use the hierarchical tree in SQL Developer to view code from your triggers. You can right click on an item to Edit the code or to Compile the object if it becomes invalid.



Section K: Database Triggers

Lesson: Viewing Stored Trigger Errors

Jump to TOC

USER_ ERRORS

Unlike procedures, functions, and packages, triggers do not display any errors in the code through the 'show errors' command. To view trigger errors during creation, select from the view USER_ERRORS.

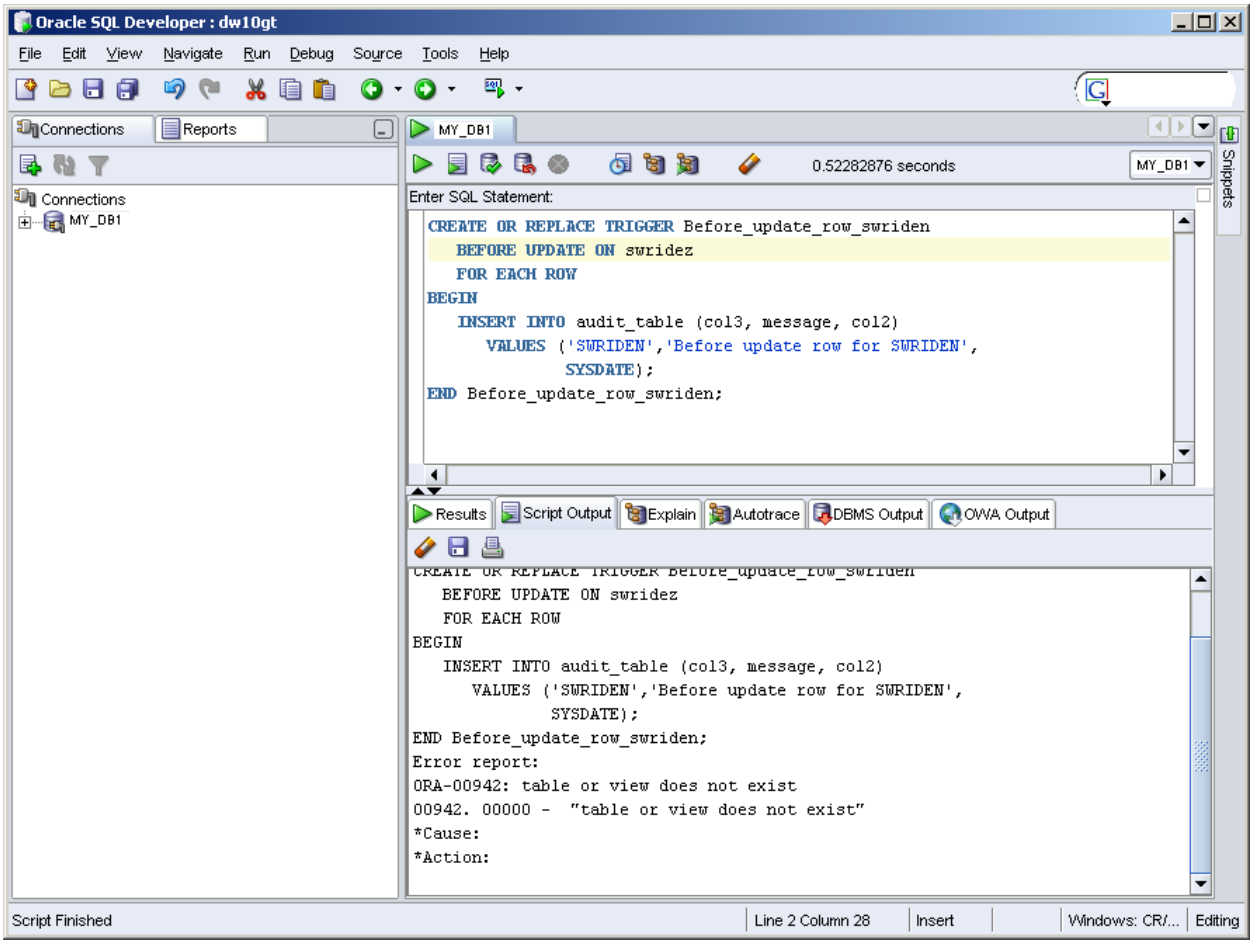
```
SQL> DESC user_errors
 Name                           Null?    Type
 ------------------------------ -------- ----
 NAME                           NOT NULL VARCHAR2(30)
 TYPE                                    VARCHAR2(12)
 SEQUENCE                       NOT NULL NUMBER
 LINE                           NOT NULL NUMBER
 POSITION                       NOT NULL NUMBER
 TEXT                           NOT NULL VARCHAR2(4000)
 ATTRIBUTE                               VARCHAR2(9)
 MESSAGE_NUMBER                          NUMBER

SQL> SELECT text from user_errors
      WHERE name = 'UPDATE_SWRIDEN';
```

Section K: Database Triggers

Lesson: Viewing Stored Trigger Errors (continued)

Jump to TOC

SQL Developer will show you the errors when you try and compile a trigger without having to select from the user_errors table.



Section K: Database Triggers

Lesson: Remove Triggers

Jump to TOC

Remove Triggers

Triggers can either be removed permanently or temporarily. You may want to temporarily turn off a trigger while doing bulk activities like data loads or conversions and then re-enable it for normal transaction processing. Note, however, that if you disable a trigger that trigger remains disabled for *all* transactions (bulk and online) until that trigger is re-enabled.

Drop a trigger

To permanently remove a trigger, use the following syntax:

```
DROP TRIGGER triggername;
```

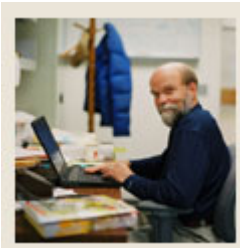
Disable/Enable a trigger

To temporarily disable a trigger and then re-enable at a later time. The syntax is:

```
ALTER TRIGGER triggername [ DISABLE | ENABLE ];
```

Examples:

```
ALTER TRIGGER update_swriden DISABLE;
ALTER TRIGGER update_swriden ENABLE;
```



Section K: Database Triggers

Lesson: Order of Trigger Firing

Jump to TOC

Order

1. Execute the BEFORE statement-level trigger, if present.

2. For each row affected by the statement:
   - Execute the BEFORE row-level trigger, if present
   - Execute the statement
   - Execute the AFTER row-level trigger, if present

3. Execute the AFTER statement-level trigger, if present.

Example

```
CREATE OR REPLACE TRIGGER Before_update_swriden
   BEFORE UPDATE ON swriden
BEGIN
   INSERT INTO audit_table (audit_id, table_name,
      message, activity_date)
   VALUES (audit_seq.NEXTVAL, 'SWRIDEN',
      'Before update statement', SYSDATE);
END Before_update_swriden;
/

CREATE OR REPLACE TRIGGER After_update_swriden
   AFTER UPDATE ON swriden
BEGIN
   INSERT INTO audit_table (audit_id, table_name,
      message, activity_date)
   VALUES (audit_seq.NEXTVAL, 'SWRIDEN',
      'After update statement', SYSDATE);
END After_update_swriden;
/

CREATE OR REPLACE TRIGGER Before_update_row_swriden
   BEFORE UPDATE ON swriden
   FOR EACH ROW
BEGIN
   INSERT INTO audit_table (audit_id, table_name,
      message, activity_date)
   VALUES (audit_seq.NEXTVAL, 'SWRIDEN',
      'Before update row '||:old.ROWID, SYSDATE);
END Before_update_row_swriden;
/

CREATE OR REPLACE TRIGGER After_update_row_swriden
   AFTER UPDATE ON swriden
   FOR EACH ROW
BEGIN
   INSERT INTO audit_table (audit_id, table_name,
      message, activity_date)
   VALUES (audit_seq.NEXTVAL, 'SWRIDEN',
      'After update row '||:old.ROWID, SYSDATE);
END After_update_row_swriden;
/
```

Subsequent update

Suppose an update occurs on the SWRIDEN table that affects three rows. If we select from the audit_table, we would get the following results:

```
     SELECT table_name, message, activity_date
     FROM audit_table
     ORDER BY audit_id;

TABLE_NAME MESSAGE                                  ACTIVITY_
---------- ---------------------------------------- ---------
SWRIDEN    Before update statement                  29-OCT-07
SWRIDEN    Before update row AAAD7oAAEAAAAL3AAP     29-OCT-07
SWRIDEN    After update row AAAD7oAAEAAAAL3AAP      29-OCT-07
SWRIDEN    Before update row AAAD7oAAEAAAAL3AAQ     29-OCT-07
SWRIDEN    After update row AAAD7oAAEAAAAL3AAQ      29-OCT-07
SWRIDEN    Before update row AAAD7oAAEAAAAL3AAR     29-OCT-07
SWRIDEN    After update row AAAD7oAAEAAAAL3AAR      29-OCT-07
SWRIDEN    After update statement                   29-OCT-07

8 rows selected.
```

Section K: Database Triggers

Lesson: Instead-of Triggers

Jump to TOC

Definition

Instead-of triggers are created against views instead of tables. It allows table data to be modified through issuing DML against a view. The trigger fires and updates the underlying table(s) INSTEAD-OF updating the view itself.

Limitations

They cannot be used against complex views that:

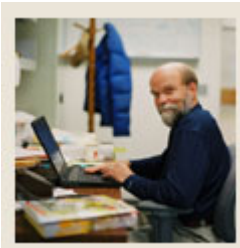
- Use the DISTINCT operator
- Use aggregate functions
- Contain a GROUP BY, CONNECT BY or START WITH clause
- Contain a subquery in a SELECT list
- Contain a set operator such as UNION, MINUS or INTERSECT

May not always be available on views that join multiple tables.

May conflict with any FGAC row-level security policies defined on the view.

Example

```
CREATE OR REPLACE TRIGGER swvtele_trig
INSTEAD OF UPDATE ON swvtele
FOR EACH ROW
BEGIN
   UPDATE swraddr
      SET swraddr_phone_area = substr(:new.swvtele_phone,2,3),
          swraddr_phone_number = substr(:new.swvtele_phone,
                                        instr(:new.swvtele_phone,'-')-3,3)
                                 ||
                                        substr(:new.swvtele_phone,
                                        instr(:new.swvtele_phone,'-') +1)
              WHERE swraddr_pidm = :new.swvtele_pidm;
END;
/
```



Section K: Database Triggers

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

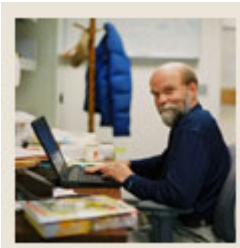
Create a sequence.

Exercise 2

Test the sequence by selecting the first value.

Exercise 3

Create a database trigger on SWBPERS. For each update statement, insert into the temp table a value from the above sequence, the current date, and the user making the update. Write a statement to update a row in the SWBPERS table and view the results of the trigger in the temp table (don't forget to commit your update).



Section K: Database Triggers

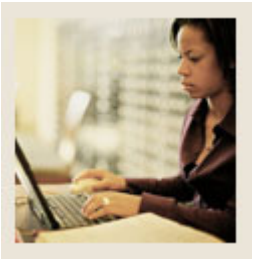
Lesson: Self Check (Continued)

Jump to TOC

Exercise 4

Create a database trigger on the SWRIDEN table. For every row inserted, check to see if a current row exists (change_ind is null). Update the original current row so that the change indicator is an *I* (ID change) or an *N* (name change), depending on the type of change.

Write an insert statement for the SWRIDEN table that results in a change to an existing record's ID or Name (do not insert a record for a new person). After committing your insert, check the table to see if the previous record for that PIDM was updated correctly.



Section L: File Input/Output

Lesson: Overview

Jump to TOC

Introduction

DBMS_OUTPUT allows you to print information to the screen. Although you could spool that information and create basic log files, it was not intended for reporting purposes. However, since file input and output is a basic need, Oracle created the package UTL_FILE, which is available in PL/SQL 2.3 and higher.

Objectives

- Input/output environments
- Steps to open, read to, write from, and close files

Section contents

Section L: File Input/Output

Lesson: Input/Output Environments

Jump to TOC

Setup

Files can only be written to the server where the database resides. There are two ways to setup the environment for writing files to disk. The first method involves setting a variable in the INIT.ORA (or SPFILE.ora) that contains a directory or set of directories where files can be written. The second method involves using Oracle directory objects. Both methods are explained below.

INIT.ORA

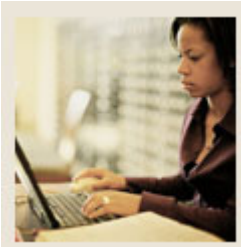
This is an older method of telling Oracle the directories where the database can read and write data. One or more directories can be added to the variable, separated by commas.

```
UTL_FILE_DIR = /y/banner/home, /x/temp
```

If the operating system is case sensitive, then you must specify the directories in the correct case.

Changing or adding directories required that the database be shutdown and restarted.

While this method is still supported it represents a security risk as any user who can log into the database can find out which directories are available for reading and writing. The second method below is the new and preferred method of defining directories for reading and writing files.



Directory Objects

A directory object is a database object with a name or alias that points to an actual directory on the database server. Each directory object can point to a different directory and there is no limit on the number of directory objects you can create.

Syntax

```
CREATE OR REPLACE DIRECTORY <internal_name> AS <full path>;
```

The path should not contain a trailing slash. Be sure to enter the directory path in the correct case for your operating system as directories are case sensitive.

```
SQL> CREATE OR REPLACE DIRECTORY my_dir AS '/u01/app/sghe/inb/my_files';
```

Security

The CREATE ANY DIRECTORY privilege must be granted to a user before they can create a directory object.

Although multiple database users can create directory objects, all directory objects are owned by the SYS user. When a directory object is created, the user who created the object is automatically granted read and write privileges to that directory.

Specific READ and/or WRITE privileges can be granted on directory objects to any number of users, making it more secure than UTL_FILE_DIR which is available to all database users.

Data Dictionary

To view available directory objects check the ALL_DIRECTORIES VIEW (there is no user_directories view as all directories are owned by the SYS or SYSTEM user).

To view who has permissions on a directory check the ALL_TAB_PRIVS which contains privileges on tables, stored PL/SQL objects, views, and directories.

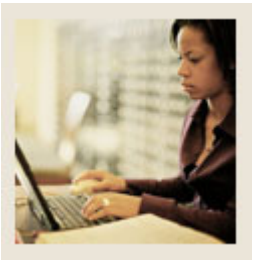
Oracle user

The File I/O operations that are performed by UTL_FILE will be done as the Oracle user. The Oracle user is the owner of the files that are used to run the database. Therefore, the Oracle user must have rights to access the directory that is referenced in the UTL_FILE call.

Make sure that all directories referenced by UTL_FILE_DIR or DIRECTORY objects have the correct permissions to allow the Oracle owner to write to those directories.

Validation

When the UTL_FILE_DIR parameter is set or a DIRECTORY object is created, the database does not check to see whether the path exists, is valid, and/or has the correct permissions. You can define a path in the database and create it later. However, if you attempt to read or write to a directory that either does not exist or cannot be written to by the Oracle software owner, you will receive a run time error.



Section L: File Input/Output

Lesson: UTL_FILE Package

Jump to TOC

Description

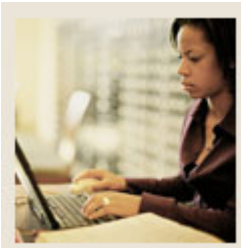
The UTL_FILE package is a built-in package that allows the reading and writing of operating system files. There are numerous procedures and functions in the package; many of which will be described below.

All file operations are done through what is called a file handle or a datatype called FILE_TYPE defined as part of the package definition. This is essentially a pointer to a file.

The file must be opened before it can be written to or read from. When writing, the package will create a file with the name specified if one does not exist.

If using directories, remember a directory object is a data dictionary object and data dictionary objects are stored in upper case in the data dictionary. When passing the directory name to any of the functions or procedures in the UTL_FILE package, make sure you pass it in upper case.

Programming tip: Don't make the user guess which case they need to use when passing parameters. Handle case sensitivities inside the PL/SQL program.



Section L: File Input/Output

Lesson: Open and Close Files

Jump to TOC

FOPEN

Opens a file for input or output and returns a pointer to the file. A file can be opened for input only or output only at any time.

```
FUNCTION FOPEN(location IN VARCHAR2,
               Filename IN VARCHAR2,
               Open_mode IN VARCHAR2)
RETURN FILE_TYPE;
```

Parameters

Parameter	Type	Description
Location	VARCHAR2	• Actual full path if using UTL_FILE_DIR. • Directory name or alias if using directories. • Case sensitive.
Filename	VARCHAR2	Name of file to be opened. If the mode is 'w', any existing file is overwritten.
Open_mode	VARCHAR2	Mode to be used. • r – Read • w – Write • a – Append
Return value	UTL_FILE.FILE_TYPE	File handle.

FCLOSE

Closes a file. When you are finished using a file, you should always close it. This will free up resources.

```
PROCEDURE FCLOSE(file_handle IN  OUT FILE_TYPE);
```

IS_OPEN

Returns TRUE if the specified file is open, and FALSE if not.

```
FUNCTION IS_OPEN(file_handle IN FILE_TYPE)
   RETURN BOOLEAN;
```

FCLOSE_ALL

Closes all open files. Intended for cleanup purposes.

```
PROCEDURE FCLOSE_ALL;
```

Note: FCLOSE_ALL will close all files and free up the resources. However, it does not mark the files as closed, so IS_OPEN will still return TRUE after an FCLOSE_ALL.

Section L: File Input/Output

Lesson: File Output

Jump to TOC

Output procedures

There are five procedures that you can use to output lines. The procedures are similar to the ones used in the DBMS_OUTPUT package.

PUT

Outputs the string to the file. Does not append a newline character.

```
PROCEDURE PUT(file_handle IN FILE_TYPE,
              Buffer      IN VARCHAR2);
```

NEW_LINE

Writes one or more newline characters to the file. The character is dependent upon the operating system.

```
PROCEDURE NEW_LINE(file_handle IN FILE_TYPE,
                   Lines       IN NATURAL := 1);
```

PUT_LINE

Outputs a string to the file including a newline character.

```
PROCEDURE PUT_LINE(file_handle IN FILE_TYPE,
                   Buffer      IN VARCHAR2);
```

PUTF

Puts a formatted string to the file. Is a limited version of the C version of fprintf().

```
PROCEDURE PUTF(file_handle IN FILE_TYPE,
  Format IN VARCHAR2,
               arg1   IN VARCHAR2 DEFAULT NULL,
               arg2   IN VARCHAR2 DEFAULT NULL,
               arg3   IN VARCHAR2 DEFAULT NULL,
               arg4   IN VARCHAR2 DEFAULT NULL,
               arg5   IN VARCHAR2 DEFAULT NULL);
```

FFLUSH

Forces the buffer to be immediately written to the specified file.

For PUT, PUT_LINE, PUTF, and NEW_LINE, data output is normally buffered instead of being immediately written to a file. Once the buffer is full, the data is written. FFLUSH forces the buffer to be written to the file.

```
PROCEDURE FFLUSH(file_handle IN FILE_TYPE);
```

GET_LINE

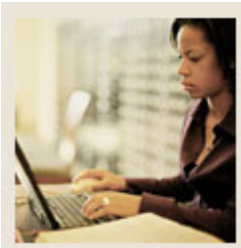
GET_LINE is used to read from a file.

```
PROCEDURE GET_LINE(file_handle IN FILE_TYPE,
                   Buffer      OUT VARCHAR2);
```

After the last line is read from the file, the next GET_LINE will raise the NO_DATA_FOUND exception.

Example

```
CREATE OR REPLACE PROCEDURE READ_FILE
/* Reads from Filename, and sends the contents to the screen. */
(pi_filedir IN VARCHAR2, pi_filename IN VARCHAR2) IS
   lv_filehandle  UTL_FILE.FILE_TYPE;
   lv_buffer_line VARCHAR2(2000);
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   lv_filehandle := UTL_FILE.FOPEN(pi_filedir,pi_filename,'r');
   LOOP
      BEGIN
         UTL_FILE.GET_LINE(lv_filehandle,lv_buffer_line);
         DBMS_OUTPUT.PUT_LINE(lv_buffer_line);
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
             EXIT;
      END;
   END LOOP;
   UTL_FILE.FCLOSE(lv_filehandle);
END READ_FILE;
/
```



Section L: File Input/Output

Lesson: Error Handling

Jump to TOC

Handling exceptions

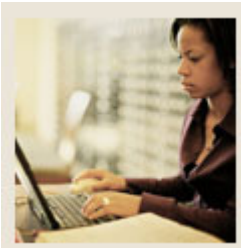
EXCEPTION	RAISED WHEN	RAISED BY
INVALID_PATH	Directory or filename is invalid or is not accessible.	FOPEN
INVALID_MODE	Invalid string specified for the file mode.	FOPEN
INVALID_FILEHANDLE	The file handle does not specify an open file.	FCLOSE, GET_LINE, PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH
INVALID_OPERATION	The file could not be opened. This could be raised because of operating system permissions. Also is raised when attempting to read from a write file, or write to a read file.	GET_LINE, PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH
READ_ERROR	An operating system error occurred as the file was read.	GET_LINE
WRITE_ERROR	An operating system error occurred during a write operation.	PUT, PUT_LINE, NEW_LINE, FFLUSH, FCLOSE, FCLOSE_ALL
INTERNAL_ERROR	An unspecified internal error occurred.	All functions
NO_DATA_FOUND	The end of file was reached during a read.	GET_LINE
VALUE_ERROR	The input line is too large for the buffer line size.	GET_LINE

1

[1] Urman, Scott. ORACLE8 PL/SQL Programming. Berkeley: Osborne McGraw-Hill., 1997.

Example

```
CREATE OR REPLACE PROCEDURE READ_WRITE_FILE
/* Reads from the in_file, and writes the data to the out_file in
double-spaced format. */
(pi_filedir IN VARCHAR2, pi_in_file IN VARCHAR2,
 pi_out_file IN VARCHAR2) IS
   lv_sourcehandle UTL_FILE.FILE_TYPE;
   lv_outhandle    UTL_FILE.FILE_TYPE;
   lv_buffer_line  VARCHAR2(2000);
BEGIN
   lv_sourcehandle := UTL_FILE.FOPEN(pi_filedir,pi_in_file,'r');
   lv_outhandle := UTL_FILE.FOPEN(pi_filedir, pi_out_file, 'w');
   LOOP
      BEGIN
         UTL_FILE.GET_LINE(lv_sourcehandle,lv_buffer_line);
         UTL_FILE.NEW_LINE(lv_outhandle);
         UTL_FILE.PUT_LINE(lv_outhandle,lv_buffer_line);
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
              EXIT;
      END;
   END LOOP;
   UTL_FILE.FCLOSE(lv_sourcehandle);
   UTL_FILE.FCLOSE(lv_outhandle);
EXCEPTION
   WHEN UTL_FILE.INVALID_OPERATION THEN
      UTL_FILE.FCLOSE_ALL;
      RAISE_APPLICATION_ERROR(-20061, 'Invalid Operation');
   WHEN UTL_FILE.INVALID_FILEHANDLE THEN
      UTL_FILE.FCLOSE_ALL;
      RAISE_APPLICATION_ERROR(-20062, 'Invalid File');
   WHEN UTL_FILE.WRITE_ERROR THEN
      UTL_FILE.FCLOSE_ALL;
      RAISE_APPLICATION_ERROR(-20063, 'Write Error');
   WHEN OTHERS THEN
      UTL_FILE.FCLOSE_ALL;
      RAISE;
END READ_WRITE_FILE;
/
```



Section L: File Input/Output

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

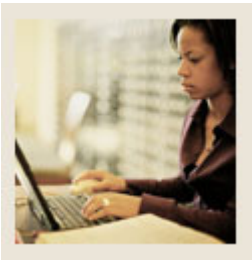
Create a directory object in the database that points to a directory supplied by the instructor. To ensure each class user creates a unique directory name include your initials in the name of the directory.

Exercise 2

Create a package called MY_TOOLS containing a stored procedure called DISPLAY_SOURCE. The procedure should accept the parameters of a directory name, output file name, and subprogram name. The DISPLAY_SOURCE procedure will retrieve the source code from USER_SOURCE for the program passed in as the subprogram name and write it to a file.

Execute the packaged procedure, and passing the directory you created above, a file name such as <Your Initials>_ex2.txt (e.g. abc_ex2.txt), and the name of one of the stored programs you created in class (e.g. CALC_AMT_OWED, ACCOUNT, INSERT_ERRORS).

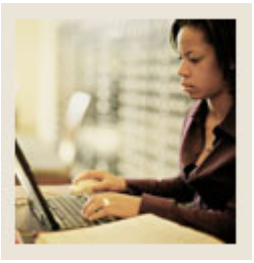
Exercise 2 (additional space)



Exercise 3

Create a second stored procedure called display_source within the MY_TOOLS package (overloading the package). The procedure should only accept the parameters of a directory name and file name. It will:

- Retrieve the source of all subprograms within your schema from USER_SOURCE and write the output to a directory and file as specified by your instructor.
- Test the procedure by executing it and passing in the name of one of the procedures or functions you created previously.
- Make this version case insensitive by handling the case restrictions inside the procedure.



Exercise 4

Create a third stored procedure within the package MY_TOOLS called LOG_ERROR. The procedure should write error messages to a log file.

- The parameters passed should be a program name and error message.
- The log file should be named the program name parameter concatenated with '.log', and in a directory created in Exercise 1.
- The procedure should write the current system time and the error message to the log file.
- Call the log_error procedure by forcing an error. Below is a sample of forcing an error:

```
DECLARE
   lv_dummy varchar2(1);
BEGIN
   SELECT '12345' INTO LV_DUMMY
     FROM DUAL;
EXCEPTION
   WHEN OTHERS THEN
      DECLARE
         lv_SQLERRM VARCHAR2(200) := SUBSTR(SQLERRM,1,200);
      BEGIN
         My_tools.error_log('<DIR_NAME>','myprogram',
            lv_sqlerrm);
      END;
END;
/
```

Exercise 4 (cont.)
(more space, if necessary)

Section M: Communicating Across Sessions

Lesson: Overview

Jump to TOC

Introduction

Two PL/SQL packages allow you to communicate across sessions that are connected to the same database. The two packages are DBMS_PIPE and DBMS_ALERT.

Objectives

Upon completion of this section, each attendee will be able to

- use the package DBMS_PIPE to send messages across sessions to other users
- use the package DBMS_ALERT to send notifications across sessions to other users.

Section contents

Section M: Communicating Across Sessions

Lesson: DBMS_PIPE

Jump to TOC

Purpose

The package DBMS_PIPE allows you to send messages across sessions to other users. The message may consist of one or more string variables packed together. The user who reads the message must know how many strings were packed together to properly unpack them.

This methodology is used in BANNER for communication between the database and external C and COBOL programs. Parameters may be packed into a message and the C or COBOL program would know how many parameters it expects and unpacks the message into the approprite parameter pieces.

When using a pipe name in all functions and procedures, those pipe names are case sensitive.

Section M: Communicating Across Sessions

Lesson: Public vs. Private Pipes

Jump to TOC

Public pipes

A pipe is implicitly created when a message is sent using an unknown pipe. This automatically creates a public pipe. A public pipe allows anyone to receive the message who has the grant and who knows the pipe name.

Private pipes

You may want to create private pipes, which will only allow the user who created the pipe to read from it. To explicitly create a pipe, use the procedure CREATE_PIPE.

CREATE_PIPE syntax

```
  CREATE_PIPE (pipename    IN VARCHAR2,
               maxpipesize IN INTEGER DEFAULT 8192,
               Private     IN BOOLEAN DEFAULT TRUE)
  RETURN INTEGER;
```

Example

```
  DECLARE
     lv_status INTEGER;
  BEGIN
     lv_status := DBMS_PIPE.CREATE_PIPE('MY_PIPE',3000,TRUE);
     DBMS_OUTPUT.ENABLE (20000);
     IF lv_status = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Successfully created pipe.');
     ELSE
        DBMS_OUTPUT.PUT_LINE('Could not create pipe.');
     END IF;
  END;
/
```

PACK_MESSAGE syntax:

Place a message into a buffer. The buffer is not written to the pipe until the SEND_MESSAGE program is called.

```
DBMS_PIPE.PACK_MESSAGE (item IN VARCHAR2 | NCHAR | NUMBER | DATE);
```

SEND_MESSAGE syntax:

Send all components created with the PACK_MESSAGE program to the pipe as a single message.

```
DBMS_PIPE.SEND_MESSAGE (pipename IN VARCHAR2,
                        timeout IN INTEGER DEFAULT MAXWAIT,
                        maxpipesize IN INTEGER DEFAULT 8192)
          RETURN INTEGER;
```

- *timeout* is the time in seconds to wait before returning
  - If *timeout* is not specified, the package constant DBMS_PIPE.MAXWAIT is used:

```
maxwait constant integer := 86400000; /* 1000 days */
( 60 seconds x 60 minutes x 24 hours x 1000 days )
```

RECEIVE MESSAGE syntax:

Take a message from a pipe and places it in an internal buffer. If there were multiple pieces packed into a message they should be unpacked into the correct number of pieces in accordance with how the message will be processed.

```
DBMS_PIPE.RECEIVE_MESSAGE (pipename IN VARCHAR2,
                           timeout IN INTEGER DEFAULT maxwait)
          RETURN INTEGER;
```

- *timeout* is the time in seconds to wait before returning
  - If *timeout* is not specified, the package constant DBMS_PIPE.MAXWAIT is used:

```
maxwait constant integer := 86400000; /* 1000 days */
( 60 seconds x 60 minutes x 24 hours x 1000 days )
```

IMPORTANT: if you do not specify a wait time, the process will wait up until 1000 days or until the database is shut down for a message. This may appear to your program as if it is 'hanging'; however, it is just waiting for a message to appear.

UNPACK_MESSAGE syntax:

Break a message received by RECEIVE_MESSAGE into its associated components. The receiving program should know how many pieces were packed so it knows how many to unpack.

```
DBMS_PIPE.UNPACK_MESSAGE(item IN VARCHAR2 | NCHAR | NUMBER | DATE);
```

Section M: Communicating Across Sessions

Lesson: DBMS_PIPE - Example

Jump to TOC

Example

```
  DECLARE
      lv_status       INTEGER;
      lv_username     VARCHAR2(30);
      lv_current_date VARCHAR2(11);
      lv_message      VARCHAR2(50);
   BEGIN
      SELECT USER, TO_CHAR(SYSDATE, 'DD-MON-YYYY'),'This is a test'
        INTO lv_username, lv_current_date, lv_message
        FROM DUAL;
      DBMS_PIPE.PACK_MESSAGE(lv_username);
      DBMS_PIPE.PACK_MESSAGE(lv_current_date);
      DBMS_PIPE.PACK_MESSAGE(lv_message);
      lv_status := DBMS_PIPE.SEND_MESSAGE('MY_PIPE_NAME');
      IF lv_status <> 0 THEN
         DBMS_OUTPUT.ENABLE(20000);
         DBMS_OUTPUT.PUT_LINE('Could not send message');
      END IF;
   END;
/
-------------------------------------------------------------------------------------
  DECLARE
     lv_var_1    VARCHAR2(100);
     lv_var_2    VARCHAR2(100);
     lv_var_3    VARCHAR2(100);
     lv_status   INTEGER;
  BEGIN
     DBMS_OUTPUT.ENABLE(20000);
     lv_status := DBMS_PIPE.RECEIVE_MESSAGE('MY_PIPE_NAME');
     IF lv_status = 0 THEN
        DBMS_PIPE.UNPACK_MESSAGE(lv_var_1);
        DBMS_PIPE.UNPACK_MESSAGE(lv_var_2);
        DBMS_PIPE.UNPACK_MESSAGE(lv_var_3);
       DBMS_OUTPUT.PUT_LINE(lv_var_1||' '||lv_var_2||' '||lv_var_3);
      ELSE
        DBMS_OUTPUT.PUT_LINE('Could not receive message');
     END IF;
  END;
/
```

Section M: Communicating Across Sessions

Lesson: Remove a Pipe

Jump to TOC

REMOVE_ PIPE

Pipes created explicitly with the CREATE_PIPE function can only be removed with the REMOVE_PIPE function. Shutting down the database clears or removes all open pipes.

Syntax

```
FUNCTION REMOVE_PIPE(pipename IN VARCHAR2)
  RETURN INTEGER;
```

Zero will be returned if the pipe is successfully removed, or if the pipe did not exist in the first place.

Example

```
DECLARE
  lv_status INTEGER;
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   lv_status := DBMS_PIPE.REMOVE_PIPE('MY_PIPE');

   IF lv_status = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Successfully removed pipe.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Could not remove pipe.');
   END IF;
END;
/
```

PURGE

To clear out the contents of a pipe, use the procedure PURGE.

Syntax

```
PROCEDURE PURGE (pipename IN VARCHAR2);
```

Example

```
BEGIN
   DBMS_PIPE.PURGE('MY_PIPE');
END;
/
```

Section M: Communicating Across Sessions

Lesson: DBMS_ALERT

Jump to TOC

Introduction

The package DBMS_ALERT allows you to send messages when transactions are committed. If you want to take some action when a transaction is committed, you can signal an alert and another process can pick up that alert and take action based on the transaction that was committed.

SIGNAL call

The SIGNAL call signals an alert in the data dictionary. The alert is not actually registered until the transaction containing SIGNAL commits; if there is a rollback, the alert is ignored.

Section M: Communicating Across Sessions

Lesson: Sending Alerts

Jump to TOC

SIGNAL procedure

The SIGNAL procedure puts an entry in the dbms_alert_info data dictionary table. When the transaction commits, the state of the alert is changed from "not signaled" to "signaled".

Syntax

```
PROCEDURE SIGNAL (name IN VARCHAR2,
                  message IN VARCHAR2);
```

Parameters

- *Name* is the name of the alert to be signaled
  - Maximum length 30 characters
  - Not case sensitive
  - Names beginning with ORA$ are reserved by Oracle
- *Message* is the message content
  - Maximum length 1800 bytes

Multiple sessions

Only one session can signal an alert at a time; if multiple sessions signal the same alert, the first session will block subsequent sessions.

Alert messages are received by all sessions that are waiting for them. If no sessions are waiting, the first session that waits will receive it immediately.

Section M: Communicating Across Sessions

Lesson: Receiving Alerts

Jump to TOC

Introduction

Sessions will receive the alerts for which they have been registered.

Registering

The REGISTER procedure registers a session's interest in an alert, which allows the session to receive that alert.

Syntax

```
PROCEDURE REGISTER (name IN VARCHAR2);
```

Parameters

- *name* is the name of the alert

Notes

- A session can register for an unlimited number of alerts.
- Registering does not cause session blocking.
- Sessions remain registered until they disconnect from the database, or until the REMOVE procedure is called

WAITONE

The WAITONE procedure waits for a specified alert.

When the alert is signaled, or if it has already been signaled, it will return with a status of 0. If the alert is not signaled, WAITONE will 'pause' the receiving session until the alert is signaled or the timeout value is reached.

If multiple messages are passed to the alert the older ones are discarded. A WAIT command will only get the most current message.

Syntax

```
PROCEDURE WAITONE (name IN VARCHAR2,
                   message OUT VARCHAR2,
                   status OUT INTEGER,
                   timeout IN NUMBER DEFAULT MAXWAIT);
```

Parameters

- *name* is the name of the specified alert
- *message* is the message associated with the specified alert
- *status* indicates whether the alert has been received
  - *0* indicates that the alert has been received, *1* indicates a timeout
- *timeout* is the time in seconds to wait before returning
  - If *timeout* is not specified, the package constant DBMS_ALERT.MAXWAIT is used:

```
maxwait constant integer := 86400000; /* 1000 days */
( 60 seconds x 60 minutes x 24 hours x 1000 days )
```

WAITANY

The WAITANY procedure is similar to WAITONE, but waits for any registered alert rather than for a specified alert.

Syntax

```
PROCEDURE WAITANY (name OUT VARCHAR2,
                   message OUT VARCHAR2,
                   status OUT INTEGER,
                   timeout IN NUMBER DEFAULT MAXWAIT);
```

Parameters

- *name* is the name of the signaled alert
  - Unlike WAITONE, *name* is an OUT parameter, returning the name of the alert that is signaled
- *message* is the message associated with the signalled alert
- *status* indicates whether the alert has been received
  - *0* indicates that the alert has been received, *1* indicates a timeout
- *timeout* is the time in seconds to wait before returning 1
  - If *timeout* is not specified, DBMS_ALERT.MAXWAIT is used

SET_DEFAULTS

The SET_DEFAULTS procedure sets the polling interval, which is the amount of time between checks for registered alerts.

This procedure is only used when the database is running in shared mode, when a polling loop is required to check for alerts signaled by other instances.

Syntax

```
PROCEDURE SET_DEFAULTS (polling_interval IN NUMBER);
```

Parameters

- *polling_interval* is the amount of time between each check for alerts
- The default is 5 seconds

Polling loops

Under most circumstances, polling loops are not required due to Oracle's event-driven nature. Loops are required when running in shared mode, as above, or when no registered alerts have triggered WAITANY.

In the latter case, the polling interval increases automatically from 1 to 30 seconds, and SET_DEFAULTS cannot be used to set the interval manually.

Example

```
DECLARE
   lv_status INTEGER;
   lv_message VARCHAR2 (100);
BEGIN
   DBMS_OUTPUT.ENABLE (20000);
   DBMS_ALERT.waitone ('ALERT1', lv_message, lv_status, 30);
   IF lv_status = 0 THEN
      DBMS_OUTPUT.put_line (lv_message);
   ELSIF lv_status = 1 THEN
      DBMS_OUTPUT.put_line ('Timeout on alert1.');
   END IF;
END;
/
```

REMOVE

The REMOVE procedure unregisters a session's interest in an alert.

Syntax

```
PROCEDURE REMOVE (name IN VARCHAR2);
```

Parameters

- *name* is the name of the alert to be unregistered

Example

```
BEGIN
   DBMS_ALERT.REMOVE('ALERT1');
END;
/
```

REMOVEALL

The REMOVEALL procedure unregisters a session's interest in all alerts. The first call to the DBMS_ALERT package calls REMOVEALL to make sure alerts from previous sessions do not get mixed up with the current session (session numbers are cycled).

This procedure has an implied commit.

Syntax

```
PROCEDURE REMOVEALL;
```

Example

```
BEGIN
   DBMS_ALERT.REMOVEALL;
END;
```

Similarities

- DBMS_PIPE and DBMS_ALERT are both implemented as PL/SQL packages, and can thus be used from any PL/SQL execution environment.
- Both packages send messages between sessions connected to the same instance.
- In PL/SQL Version 2, pipes and alerts are the only ways to send messages to a waiting C daemon. (Oracle8 circumvents this via external procedures, which are outside the scope of this workbook.)

Differences

Pipes	Alerts
Asynchronous; Messages are sent as soon as DBMS_PIPE.SEND_MESSAGE is issued, and rollbacks will not retrieve them.	Transaction-based; Alerts are not sent until the transaction containing DBMS_ALERT.SIGNAL is committed.
If there are multiple sessions waiting for a pipe message, only one will receive it.	All sessions that are registered for an alert will receive its message when it is signaled.
The entire contents of the message buffer are sent, which may include a variety of information.	Alerts can send only a single character string.
Pipes are used for two-way communication.	Alerts are simple one-way messages.

Section M: Communicating Across Sessions

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

Create a public pipe named *TRAINx_PIPE*, where *x* is your training account number. Submit a message to the pipe.

Exercise 2

Retrieve a message from a pipe that your neighbor submitted. Are you able to receive each other's messages? (Remember, pipe names are case sensitive.)

Exercise 3

Create an alert called *TRAINx_ALERT*, where *x* is your training account number. Work with a neighbor to make sure that each of you can register and receive notification from an alert.

Section N: Dynamic SQL

Lesson: Overview

Jump to TOC

Introduction

Although stored functions and procedures can enhance modularity and flexibility, we have not yet been able to dynamically create a select statement, for example. Dynamic SQL is a powerful addition to the PL/SQL toolset.

Objectives

Upon completion of this section, each attendee will be able to

- define the steps necessary to build a dynamic query
- understand the possibilities the package provides by examining examples
- write and execute a dynamic query using a stored function.

Section contents

Section N: Dynamic SQL

Lesson: Dynamic SQL Steps

Jump to TOC

Dynamic SQL

Dynamic SQL became available within PL/SQL 2.1 (Oracle 7.1) and higher.

Dynamic SQL statements are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments.

Steps

The steps for executing a statement using DBMS_SQL are the following:

- Assign the SQL statement to a string variable.
- Parse the string (PARSE).
- Bind any input variables (BIND_VARIABLES).
- If the statement is a query, define the output variables (DEFINE_COLUMN).
- Execute the query, and fetch the results using
  - EXECUTE
  - FETCH_ROWS
  - COLUMN_VALUE
  - VARIABLE_VALUE

OPEN_ CURSOR

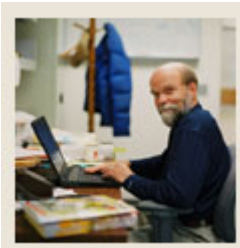
Returns a cursor ID number used to identify the context area in which the statement will be processed.

PARSE

Sends the statement to the server, where syntax is verified. If the statement is a query, the execution plan is determined.

BIND_VARIABLE

Binds a variable to a placeholder. Binding is done for input variables.



EXECUTE

For a non-query, EXECUTE will carry out the statement and return the number of rows processed. For a query, EXECUTE will determine the active set. The data is then fetched with FETCH_ROWS.

VARIABLE_VALUE

Used to determine the value of a bind variable if it is modified by the statement.

COLUMN_VALUE

Used to return the data.

CLOSE_CURSOR

Closes the cursor, and frees all resources used by the cursor.

Example 1

```
CREATE OR REPLACE FUNCTION f_validate_single (pi_owner VARCHAR2,
pi_tablename VARCHAR2, pi_column VARCHAR2, pi_value VARCHAR2)
RETURN VARCHAR2 IS
/*********************************************************
Powerful way to validate codes.  Pass in the owner, table, and column
of the validation table, along with the value you are verifying.
The function will dynamically create a select statement based upon
your parameters.  The function returns 'FALSE' if no errors (it
found the entry), and 'TRUE' if no data is found for the value you
are passing.
*********************************************************/
   lv_statement VARCHAR2(300);
   lv_cursor    INTEGER;
   lv_return    INTEGER;
   lv_sql_vers  CONSTANT INTEGER := 1; -- 1=native; 0=Vers6; 2=Vers7
BEGIN
   lv_statement := 'SELECT '||''''||'X'||''''||
                   ' FROM '||pi_owner||'.'||pi_tablename||
                   ' WHERE '||pi_column||'='||''''||pi_value||'''';
   lv_cursor := dbms_sql.open_cursor;
   dbms_sql.parse(lv_cursor,lv_statement,lv_sql_vers);
   lv_return := dbms_sql.execute(lv_cursor);
    /*Execute the SQL Command */
IF DBMS_SQL.FETCH_ROWS (lv_cursor) <> 0 THEN RETURN 'FALSE';
ELSE
   RETURN 'TRUE';
END IF;
DBMS_SQL.CLOSE_CURSOR (lv_cursor);
EXCEPTION
   WHEN OTHERS THEN
   DECLARE
      lv_err_msg VARCHAR2(207) := 'ERR - '||substr(SQLERRM,1,200);
   BEGIN
      RETURN lv_err_msg;
   END;
END f_validate_single;
/
```

Example 2

```
CREATE OR REPLACE function f_get_value (pi_select_column VARCHAR2,
                pi_where_column VARCHAR2, pi_where_value VARCHAR2)
RETURN VARCHAR2 IS
/* Creates a dynamic SQL statement which allows you to select any
column from SWRIDEN (select_column), where a particular column
(where_column) equals a certain value (where_value). */
   lv_statement VARCHAR2(800);
   lv_cursor    INTEGER;
   lv_return    INTEGER;
   lv_out_value VARCHAR2(2000);
   lv_sql_vers  CONSTANT INTEGER := 1; -- 1=native; 0=Vers6; 2=Vers7
BEGIN
   lv_cursor := dbms_sql.open_cursor;
   lv_statement := 'BEGIN
                      SELECT '||pi_select_column||
                      ' INTO :lv_out_value
                        FROM SWRIDEN
                       WHERE '||
                     pi_where_column||' = :pi_where_value;
                   END;';
   dbms_sql.parse(lv_cursor,lv_statement,lv_sql_vers);
   dbms_sql.bind_variable(lv_cursor, ':lv_out_value',
                          lv_out_value,2000);
   dbms_sql.bind_variable(lv_cursor, ':pi_where_value',
                          pi_where_value,2000);
   lv_return := dbms_sql.execute(lv_cursor);
   dbms_sql.variable_value(lv_cursor, 'lv_out_value', lv_out_value);
   DBMS_SQL.CLOSE_CURSOR(lv_cursor);
  RETURN lv_out_value;
EXCEPTION
   WHEN OTHERS THEN
   DECLARE
      lv_err_msg VARCHAR2(207) := 'ERR - '||substr(SQLERRM,1,200);
   BEGIN
      RETURN lv_err_msg;
   END;
END f_get_value;
/
```

Example

```
/* Prompts for a column name and table name, and will print the
column contents to the screen. */
DECLARE
   lv_select_column VARCHAR2(200) := '&SELECT_COLUMN';
   lv_table_name    VARCHAR2(30) := '&TABLE';
   lv_statement     VARCHAR2(800);
   lv_cursor        INTEGER;
   lv_return        NUMBER;
   lv_out_value     VARCHAR2(2000);
   lv_err_msg       VARCHAR2(200);
   lv_act_length    INTEGER;
   lv_sql_vers   CONSTANT INTEGER := 1; -- 1=native; 0=Vers6; 2=Vers7
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   lv_cursor := DBMS_SQL.OPEN_CURSOR;
   lv_statement := 'SELECT '||lv_select_column||
                   ' FROM '||LV_TABLE_NAME;
   DBMS_OUTPUT.PUT_LINE(LV_STATEMENT);
   DBMS_SQL.PARSE(lv_cursor,lv_statement,lv_sql_vers);
   DBMS_SQL.DEFINE_COLUMN(lv_cursor,1,lv_out_value,2000);
   lv_return := DBMS_SQL.EXECUTE(lv_cursor);
   LOOP
      IF DBMS_SQL.FETCH_ROWS(lv_cursor) = 0 THEN
         EXIT;
      END IF;
      DBMS_SQL.COLUMN_VALUE(lv_cursor,1,lv_out_value);
      DBMS_OUTPUT.PUT_LINE(LV_OUT_VALUE);
   END LOOP;
   DBMS_SQL.CLOSE_CURSOR(lv_cursor);
EXCEPTION
   WHEN OTHERS THEN
   DECLARE
      lv_err_msg VARCHAR2(207) := 'ERR - '||substr(SQLERRM,1,200);
   BEGIN
      DBMS_OUTPUT.PUT_LINE(lv_err_msg);
   END;
END;
/
```

Section N: Dynamic SQL

Lesson: What are all those Quotes?

Jump to TOC

Adding Quotes to Dynamic SQL

If you need quotes to appear in the results of your Dynamic SQL there are two ways of accomplishing this.

1. Add three (3) single quotes to the beginning or end of a literal string to add a single quote to the output (at the beginning or end of the literal string).

```
SELECT 'select ''' || table_name || ''', count(*) from ' ||
       table_name || ';'
  FROM user_tables
 ORDER BY table_name;

'SELECT'''||TABLE_NAME||''',COUNT(*)FROM'||TABLE_NAME||';'
```

```
select 'HIGH_MATH', count(*) from HIGH_MATH;
select 'HIGH_VERBAL', count(*) from HIGH_VERBAL;
select 'SWBADDR', count(*) from SWBADDR;
select 'SWBPERS', count(*) from SWBPERS;
select 'SWRADDR', count(*) from SWRADDR;
select 'SWRIDEN', count(*) from SWRIDEN;
select 'SWRREGS', count(*) from SWRREGS;
select 'SWRSTDN', count(*) from SWRSTDN;
select 'SWRTEST', count(*) from SWRTEST;
. . .
```

2. Add four (4) single quotes if you are appending a single quote by itself:

```
SELECT DISTINCT 'select swraddr_atyp_code, count(*) ' ||
                'from swraddr where swraddr_atyp_code = ''' ||
                swraddr_atyp_code || '''' || ';' my_sql
  FROM swraddr;

MY_SQL
----------------------------------------------------------------
select swraddr_atyp_code, count(*) from swraddr where swraddr_atyp_code = 'MA';
select swraddr_atyp_code, count(*) from swraddr where swraddr_atyp_code = 'P1';
select swraddr_atyp_code, count(*) from swraddr where swraddr_atyp_code = 'PR';
```

Section N: Dynamic SQL

Lesson: Execute Immediate

Jump to TOC

Description

Execute Immediate is a built-in command you can use to execute a dynamic SQL statement in PL/SQL. (Do not confuse this with the function execute_immediate in the dbms_hs_passthrough package.)

Any valid DML statement can be executed except a SELECT that return more than one row (SELECT…INTO is allowed). It may also be an anonymous PL/SQL block or a call to a stored program.

Bind Variables can be passed in the USING clause.

Single-row select statements will have their results returned in the variables in the INTO clause.

Syntax

```
EXECUTE IMMEDIATE dynamic_sql_string
       [INTO {define_var1 [, define_var2] ... | plsql_record}]
       [USING [IN | OUT | IN OUT] bind_arg1 [,
              [IN | OUT | IN OUT] bind_arg2] ...]
       [{RETURNING | RETURN} INTO bind_arg3 [, bind_arg4] ...];
```

Section N: Dynamic SQL

Lesson: Execute Immediate (continued)

Jump to TOC

Examples

```
    declare
        lv_sql varchar2(500);
    begin
        lv_sql := 'create table error_table (prog_name varchar2(30),
err_date date, err_msg varchar2(300))';
        EXECUTE IMMEDIATE lv_sql;
    end;
    /

    declare
        lv_sql varchar2(500);
    begin
    lv_sql := 'insert into temp values (1, ''Example 2'',
          SYSDATE, ''Hello'')';
        EXECUTE IMMEDIATE lv_sql;
    end;
    /

    declare
        lv_sql varchar2(500);
        lv_val varchar2(50) := 'Dynamic insert statement';
    begin
        lv_sql := 'insert into temp values (1,sysdate,:b1)';
        EXECUTE IMMEDIATE lv_sql USING lv_val;
    end;
    /

    declare
        lv_sql varchar2(500);
        lv_val number(9);
        lv_ret varchar2(50);
    begin
        lv_sql := 'select count(*) from swriden where swriden_pidm =
:b1';
        lv_val := 12340;
        EXECUTE IMMEDIATE lv_sql INTO lv_ret USING lv_val;
        dbms_output.put_line('Count: '|| lv_ret || ' For Pidm: ' ||
      lv_val);
    end;
```

Section N: Dynamic SQL

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

Create a procedure p_pidm_tables by running the script in your student_files folder called SectN_Ex1.sql. The procedure can be used to identify which tables and how many records per table a person has. This information may be useful in a data cleanup situation.

- The procedure accepts the parameter of pidm and schema owner.
- It retrieves tables from the view ALL_TAB_COLUMNS where the column name is like '%PIDM%'.
- Then evaluates if a row exists in each table (derived from above step) for the pidm passed in. If so, print the table name to the screen.

Execute the procedure, passing in your user ID (TRAINxx) and a PIDM from the SWRIDEN table. This will return how many records for that pidm exist in each of your tables. Run it again with a different pidm.

Exercise 2

Write an anonymous block that uses the EXECUTE IMMEDIATE built-in to execute an SQL statement to find out how many address records a person has. Prompt for a PIDM, feeding the data to a bind variable in the SQL string. Display the results of the SQL statement execution.

Section O: Optimizing Code

Lesson: Overview

Jump to TOC

Introduction

SQL has been designed to be able to easily query and manipulate the database, and as a programmer you'll find that there may be multiple ways to 'ask' the Oracle database to perform a particular task. However, although your process may be working correctly, it is important to review whether a process is working efficiently.

Optimizing code may seem to be an unnecessary step, but that view can be costly. Although a database administrator or a network person can enhance the performance of a database, we will focus on several preventative steps a developer can take to improve his/her application.

Objectives

This section will examine the following:

- Incentives for tuning
- When to tune SQL
- Optimizing statements

Section contents

Section O: Optimizing Code

Lesson: Incentives for Tuning

Jump to TOC

Incentives

- To improve interactive response time for users
- To improve batch throughput
- To ensure scalability
- To reduce system load
- To avoid hardware upgrades

Section O: Optimizing Code

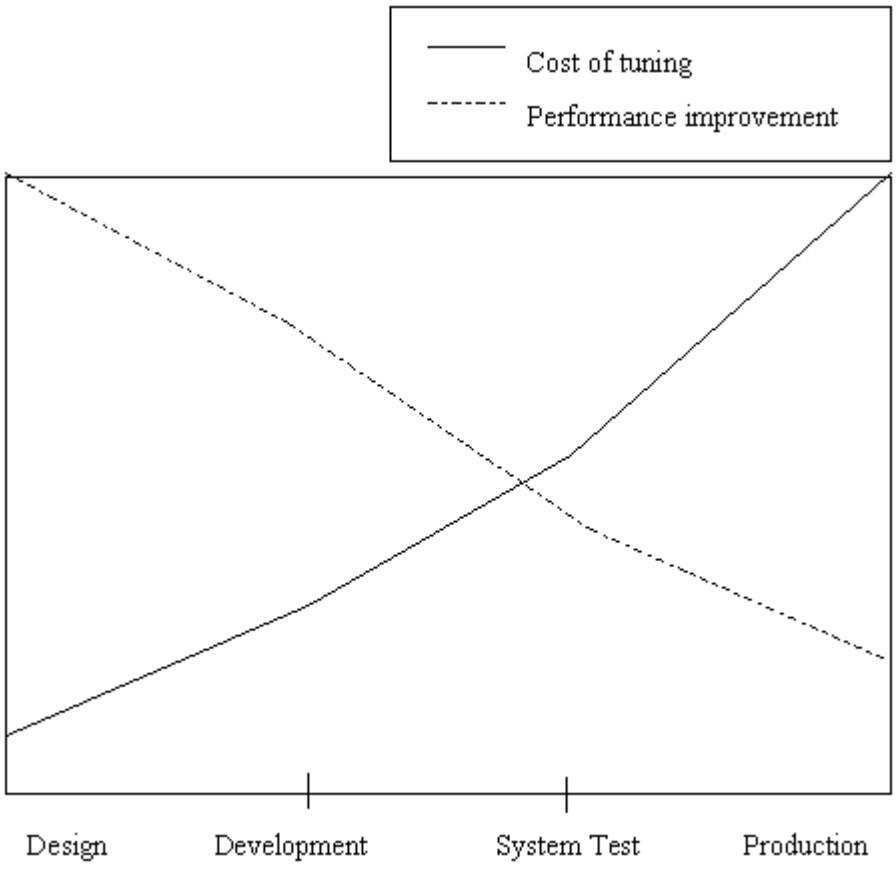
Lesson: When to Tune SQL

Jump to TOC

Timing

Ideally, SQL should be tuned as it is written. Below is a cost analysis of tuning during the life of an application.

Diagram



Section O: Optimizing Code

Lesson: Aspects of SQL Tuning

Jump to TOC

Aspects

- Tuning is an iterative process
  - Tune now for performance based on current data
  - Tune later as data changes/increases
- Measuring the performance of SQL is critical. Without measurements, we will not know whether our tuning was helpful
- A wide range of methods exist to improve performance, and it is important to understand how Oracle processes SQL to know which tool to use

The Tuning Environment

The ideal tuning environment would be one in which:

- data volumes are realistic
  - Testing/Tuning on small data volumes does not translate to large data volumes
- system performance requirements are spelled out
- data model documentation is available and is easy to understand
- validation of performance requirement is built in to the quality assurance process (what are acceptable ranges).
  - Is 30 seconds acceptable or does it need to be sub-second (Web Apps)
  - Is a 2 hour report acceptable or should it be < 30 minutes

SQL Tuning Techniques

- Rewording your SQL
- Giving Oracle explicit instructions (hints) which direct Oracle to retrieve and process the data in a particular way
- Creating or changing indexes
  - Are there indexes on frequently used columns
- Verifying existing indexes
  - Are the indexes valid (i.e. not corrupt)
- Validating table statistics (10g)
  - Do all tables have statistics
  - Are the statistics up to date

Section O: Optimizing Code

Lesson: How Oracle Processes a SQL Statement

Jump to TOC

Processing

It is important to understand how Oracle processes statements in order to know how to resolve bottlenecks. When a statement is submitted, the following occurs:

- Check syntax
- Search System Global Area (SGA) for a parsed copy of the same statement
- Search data dictionary for security privileges, synonyms, views, etc.
- Check for a saved search plan
- Calculate search plan
- Save execution plan

Section O: Optimizing Code

Lesson: The System Global Area

Jump to TOC

System Global Area (SGA)

The System Global Area can hold statements that have previously been parsed during the session. Avoiding the step of parsing can save a significant amount of time. However, SQL cannot be shared within the SGA unless it is identical.

The following statements would not be considered the same according to SGA:

```
/* Statement 1: Original Statement */
SELECT SWRIDEN_PIDM FROM SWRIDEN WHERE SWRIDEN_ID = '12341';

/* Statement 2: Different Ids */
SELECT SWRIDEN_PIDM FROM SWRIDEN WHERE SWRIDEN_ID = '12340';

/* Statement 3: Different Case */
SELECT SWRIDEN_PIDM FROM SWRIDEN WHERE swriden_id = '12341';

/* Statement 4: Different amount of white space */
SELECT SWRIDEN_PIDM
       FROM SWRIDEN
      WHERE SWRIDEN_ID = '12341';
```

Therefore, standards among programmers at your institution are important in order to take advantage of repetitive parsing.

Bind Variables

How can you avoid the case where the variables, such as ID shown above, need to change? If you use bind variables, then the bind variable references are the same, since bind variables are not substituted until a statement has been successfully parsed. Since this may mean extra effort and coding, this should only be attempted when the SQL statement will be used repetitively.
SELECT SWRIDEN_PIDM FROM SWRIDEN WHERE SWRIDEN_ID = :ID;

Note: For static application interfaces such as Oracle Forms, Pro*C, Pro*COBOL, etc., statement case, whitespace, and bind variables will always be the same for the same statements. All users execute the identical SQL statement regardless of the bind variable name or value supplied.

Table Aliases

A standard should be made for table aliases. If a single table is included in two SQL statements but under a different alias in each, the SQL statements cannot be shared even if they are otherwise identical.

Section O: Optimizing Code

Lesson: The SQL Optimizer

Jump to TOC

Criteria

When a statement is submitted, Oracle chooses the optimal execution plan, or retrieval path, for the statement. The optimizer considers the following criteria:

- The syntax you have specified for the statement
- Any conditions that the data must satisfy (the WHERE clause)
- The tables your statement will need to access
- Possible indices that can be used in retrieving data from the table
- The RDBMS version
- The current optimizer mode
- SQL statement hints
- Available object statistics (generated via the ANALYZE command)
- INIT.ORA settings

Section O: Optimizing Code

Lesson: Rule-Based vs. Cost-Based Optimization

Jump to TOC

Optimizer types

The type of optimizer Oracle uses is specified via an initialization parameter in the INIT.ORA or SPFILE.ora file.

Rule-based

The rule-based optimizer uses a predefined set of precedence rules to figure out which path it will use to access the database. These rules were developed many years ago and have not been updated by Oracle since Version 7. This method has been formally deprecated in Version 10g but is still used internally when a hint is specified for an SQL statement.

Cost-based

The cost-based optimizer (delivered in Oracle7) is more sophisticated, but requires more effort by the database administrator. Rather than relying on a set of standard rules, it requires that all the referenced tables be analyzed beforehand (except 10g), and table size makes a significant difference. Therefore, if a table has not been analyzed recently, the optimizer information may be incorrect causing a less than optimal execution path.

In 10g Oracle will analyze a table and compute new statistics on the fly as it is calculating an execution path. While this may produce a more optimal plan it takes time to compute those statistics, especially on large tables. It is the responsibility of the DBA to update those statistics while end users are not accessing the database so as not to impede performance.

Section O: Optimizing Code

Lesson: Rule-Based Optimizer Tuning

Jump to TOC

Rules to Observe

The following are some of the Rule Based Optimizer rules that help determine the execution path. These do not apply to databases running Oracle 10g or the Cost Based Optimizer in earlier versions.

Table Order

Table order is important in Rule Based Optimization. It evaluates tables from right to left. Therefore, the table that returns the fewest number of rows should be the last table in the FROM clause.

WHERE Clause Sequencing

The order in which you specify conditions in the WHERE clause has a major impact on performance. In the absence of any other information, the rule-based optimizer must use the WHERE clauses sequencing to help determine the best execution path within the database. If you are able to specify the more efficient indexed conditions early in your WHERE clause, the optimizer will be more likely to choose the most efficient execution path.

Two or More Equality Indices

When an SQL statement has two or more equality indices over different tables (Where = value) available to the execution plan, Oracle uses both indices by merging them at run-time and fetching only rows that are common to both indices. If two indices exist over the same table in a WHERE clause, Oracle ranks them. A UNIQUE index will always rank higher than a non-UNIQUE index.

Index Always Used

The Rule Based Optimizer will always use an index if it finds one that fits the situation. This is not always optimal; especially for small tables or queries that will return more than 40% of the rows from the table.

Section O: Optimizing Code

Lesson: Cost Based Optimizer Tuning

Jump to TOC

No Rules, Just Statistics

There are no pre-set rules that apply to Cost Based Optimization. Many factors are evaluated as Oracle tries to determine the most efficient path.

Some flexibility in the way the cost-based optimizer works is provided by the following optimizer mode variations, specified via the OPTIMIZER_MODE initialization parameter.

OPTIMIZER_MODE=FIRST_ROWS_n

A cost-based approach with a goal of best response time to return the first n rows (where n = 1, 10, 100, 1000)

OPTIMIZER_MODE=FIRST_ROWS

Uses a mix of costs and heuristics to find a best plan for fast delivery of the first few rows

OPTIMIZER_MODE=ALL_ROWS

A cost-based approach with a goal of best overall throughput (minimum resource use to complete the entire statement)

Timing is Everything

There is a preset amount of time that Oracle takes to determine the optimal execution path. At the end of that time period, it chooses the most efficient of the paths it calculated. For extremely complex SQL, Oracle may not have sufficient time to calculate the optimal path. In those cases, there are ways to "assist" Oracle in using the optimal path.

Section O: Optimizing Code

Lesson: Explain Plan

Jump to TOC

EXPLAIN PLAN tool

To identify the execution path for a statement, use the tool EXPLAIN PLAN.

Preparation

EXPLAIN_PLAN results detailing the execution plan go into a table called PLAN_TABLE. The PLAN_TABLE is automatically created in 10g as a global temporary table to hold the output of an EXPLAIN PLAN statement for all users. If a local PLAN_TABLE is desired, ORACLE_HOME/rdbms/admin/utlxplan.sql can be run to create a plan_table in a user's own schema before running EXPLAIN PLAN.

Example

```
EXPLAIN PLAN FOR
SELECT swriden_last_name, swriden_first_name
  FROM swriden, twraccd
 WHERE swriden.swriden_pidm = twraccd.twraccd_pidm
   AND swriden.swriden_change_ind IS NULL
   AND twraccd.twraccd_paid_date IS NULL;

Explained.
```

The "Explain Plan For…" statement does not run the SQL statement; it merely computes the execution path.

Function and examples

Executes a hierarchical query against the plan table to retrieve the execution plan.

```
SELECT RTRIM(LPAD(' ',2*LEVEL)||
  RTRIM(OPERATION)||' '||
  RTRIM(OPTIONS)||' '||
  OBJECT_NAME) QUERY_PLAN
  FROM PLAN_TABLE
CONNECT BY PRIOR ID=PARENT_ID
START WITH ID=0;

QUERY_PLAN
----------------------------------------
  SELECT STATEMENT
    MERGE JOIN
      SORT JOIN
        TABLE ACCESS FULL TWRACCD
      SORT JOIN
        TABLE ACCESS FULL SWRIDEN
```

See how adding an index can change the execution path:

```
SQL> CREATE INDEX twraccd_index on twraccd(twraccd_pidm);

SQL> EXPLAIN PLAN FOR
SELECT swriden_last_name, swriden_first_name
  FROM swriden, twraccd
 WHERE swriden.swriden_pidm = twraccd.twraccd_pidm
   AND swriden.swriden_change_ind IS NULL
   AND twraccd.twraccd_paid_date IS NULL;

Explained.


QUERY_PLAN
----------------------------------------
  SELECT STATEMENT
    NESTED LOOPS
      TABLE ACCESS FULL SWRIDEN
      TABLE ACCESS BY ROWID TWRACCD
        INDEX RANGE SCAN TWRACCD_INDEX
```

Section O: Optimizing Code

Lesson: Explain Plan (Continued)

Jump to TOC

Query Plan

Interpreting the query plan:

- When reading the query plan, the innermost statement is executed first (indented the most).
- If the COST= is blank, then the database is using the Rule Based Optimizer.
- The higher the cost, the more "expensive" or resource intensive the SQL statement is.
- Statements at the same level are executed from top to bottom.

Section O: Optimizing Code

Lesson: Autotrace

Jump to TOC

Autotrace

Autotrace is a utility that can provide an execution plan for your statement as well as execution statistics. It has an advantage over issuing the *explain plan for…*statement since it doesn't require cleaning out the plan table in between executions; nor does it require you to memorize or store the crypic SQL required to extract data from the plan table.

Requirements

For Autotrace to work the DBA needs run the following script as the user SYS: $ORACLE_HOME/sqlplus/admin/plustrce.sql. This creates a database role called PLUSTRACE that needs to be granted to the users who will be using the utility. It also requires that the user have a plan table (instructions on creating a plan table are above).

Syntax

SET AUTOTRACE ON | OFF | TRACEONLY
SET AUTOTRACE ON { EXPLAIN | STATISTICS }

Example

```
SQL> set autotrace traceonly
SQL> SELECT swriden_last_name, swriden_first_name
  2    FROM swriden, twraccd
  3   WHERE swriden.swriden_pidm = twraccd.twraccd_pidm
  4     AND swriden.swriden_change_ind IS NULL
  5     AND twraccd.twraccd_paid_date IS NULL;

18 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 819603857

--------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |    20 |  1200 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN         |         |    20 |  1200 |     7  (15)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| SWRIDEN |    10 |   380 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| TWRACCD |    18 |   396 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("SWRIDEN"."SWRIDEN_PIDM"="TWRACCD"."TWRACCD_PIDM")
   2 - filter("SWRIDEN"."SWRIDEN_CHANGE_IND" IS NULL)
   3 - filter("TWRACCD"."TWRACCD_PAID_DATE" IS NULL)

Statistics
----------------------------------------------------------
        632  recursive calls
          0  db block gets
        155  consistent gets
         15  physical reads
          0  redo size
        821  bytes sent via SQL*Net to client
        392  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
         10  sorts (memory)
          0  sorts (disk)
         18  rows processed
```

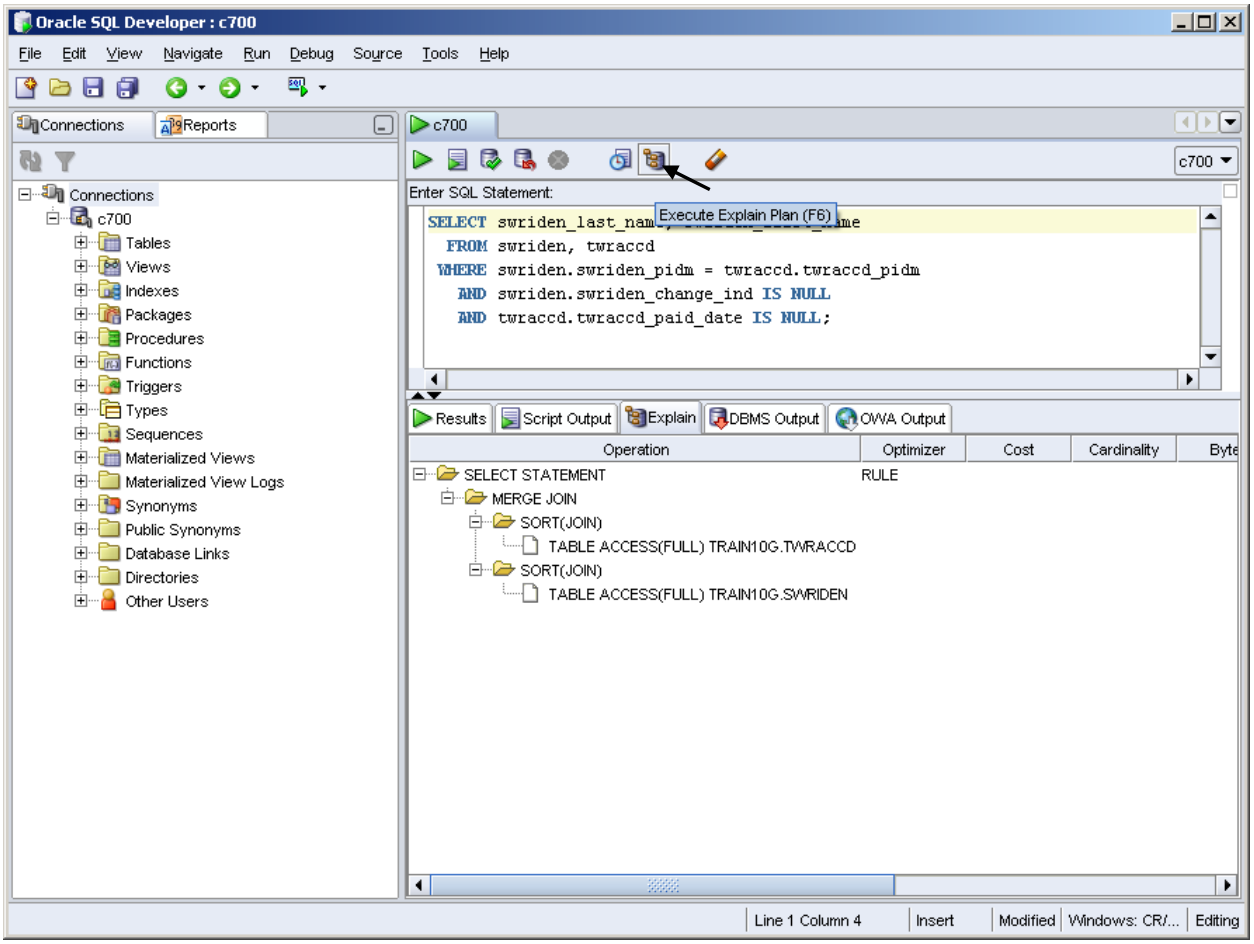
Section O: Optimizing Code

Lesson: Explain in SQL Developer

Jump to TOC

Description

SQL Developer has a quick and easy way of generating an execution plan.



Section O: Optimizing Code

Lesson: What to Watch

Jump to TOC

Problem Areas

When examining an execution plan there are a few things that you can spot and examine for potential tuning opportunities. This is only a small sample. SQL tuning takes time and practice to learn what to look for and how to fix it.

TABLE ACCESS FULL – means that it is reading each and every row in the table

- Does the table have any indexes? If not, consider creating one or more on frequently joined columns.
- Are you referencing any columns in the index(es) in your where clause? If so, is at least one of the columns the first column in a composite index?
- Is the index on a column with low cardinality (small number of unique values)?
- If there is not an index on a table using the column(s) in your join, are these columns frequently used in a join? If so, consider adding an index.
- Are there a very small number of rows in the table? If so, it may be faster to read all the rows at once.
- If there is an index that you can use and it's not being used, is the index valid? Could the index be corrupt?
- Is the column(s) of an index being used in a function (UPPER, TO_CHAR, etc)? Functions on columns will cause indexes to not be used unless a function-based index has been created.

MERGE JOIN CARTESIAN – did you forget to join a table?

SORT MERGE JOIN – (see example above)

- sorts are expensive (i.e. resource intensive)

NESTED LOOP – for each row in Table 1 get corresponding rows in Table 2 (example above after adding index)

- Make sure Table 1 returns the fewest amount of rows

Section O: Optimizing Code

Lesson: Elapsed Times

Jump to TOC

SET TIMING ON

As you tune your statements, you will want to know how much more efficient the statement is in relation to time. Use the SQL*Plus command 'SET TIMING ON' to view the elapsed time needed for a statement to be executed.

Example

```
SQL> SET TIMING ON

SQL> SELECT * from dictionary;

  TABLE_NAME                COMMENTS
  ------------------------- ----------------------------------------
  …
  ALL_OUTLINES              Synonym for USER_OUTLINES
  ALL_OUTLINE_HINTS         Synonym for USER_OUTLINE_HINTS
  USER_SQLSET_DEFINITIONS   Synonym for USER_SQLSET
  ALL_OLAP_ALTER_SESSION    Synonym for V$OLAP_ALTER_SESSION

  662 rows selected.

  Elapsed: 00:00:02.81
```

Other factors

There are other factors relating to elapsed time, including network traffic, other processes running, etc. However, a consistent improvement in elapsed time will indicate your tuning was successful.

Section O: Optimizing Code

Lesson: Self Check

Jump to TOC

Directions

Use the information you have learned in this workbook to complete this self-check activity.

Exercise 1

Set the timing on in your SQL*Plus session and execute the following select statement (if using SQL Developer, just click on the Execute Explain Plan button):

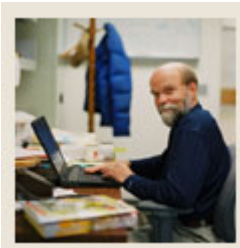
```
SELECT SWRADDR_STREET_LINE1, SWRADDR_ZIP, SWRADDR_CITY,
SWRADDR_STAT_CODE
  FROM SWRIDEN, SWRADDR
 WHERE SWRIDEN.SWRIDEN_PIDM = SWRADDR.SWRADDR_PIDM;
```

Note the amount of time it took for the statement to execute.

Explain the plan for the select statement. (If using SQL Developer, the Explain Tab should have been activated when you pressed the Execute Explain Plan button.)

Exercise 2

Are indices being used, or is Oracle executing a full-table scan? If no indices are being used, then create an index for SWRADDR. Rerun Explain Plan and view the results to make sure your indices are being used. Rerun the Select statement and compare the execution time. Have you improved the performance of the statement?



Section P: Appendix

Lesson: Overview

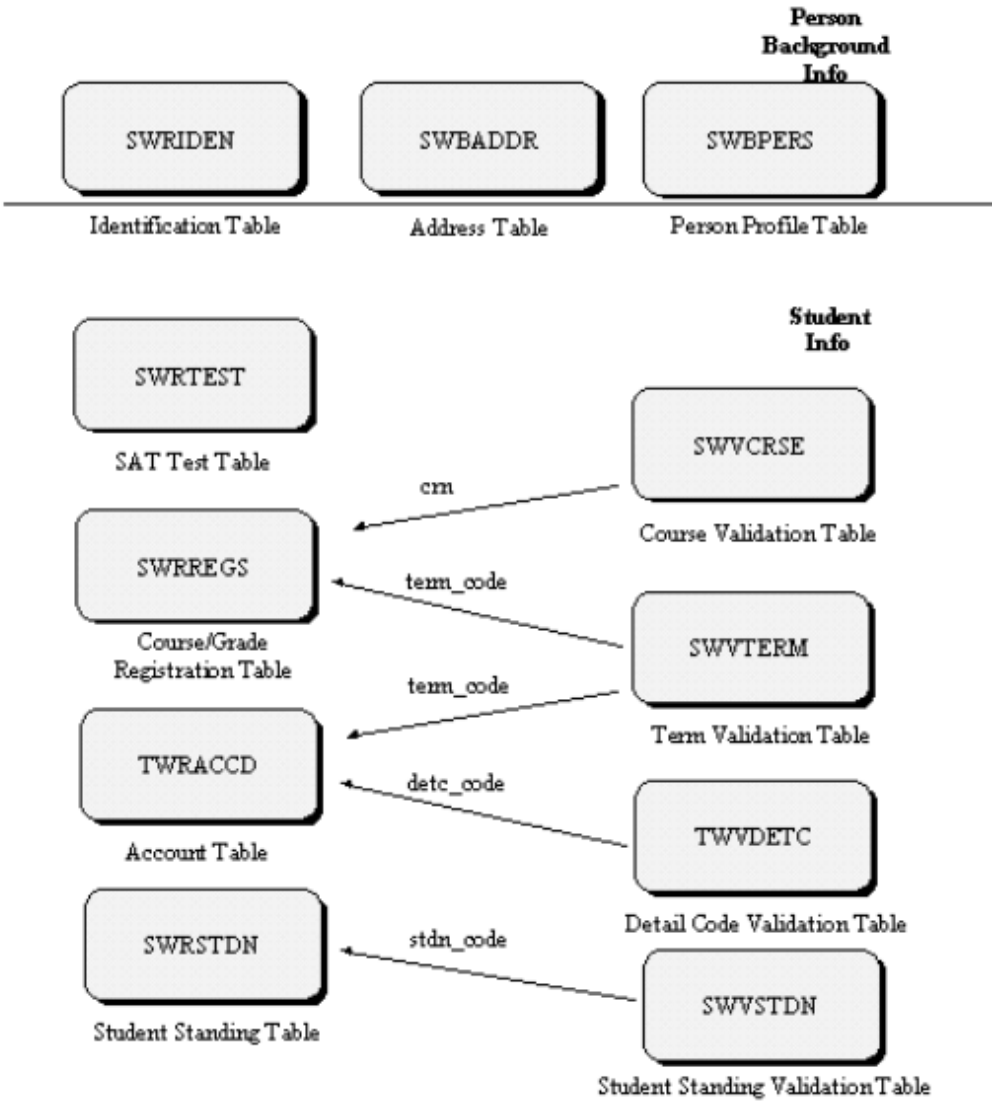
Jump to TOC

Introduction

This section contains miscellaneous topics of interest.

Section contents

Diagram


Person Background Info
SWRIDEN
SWBADDR
SWBPERS
Identification Table
Address Table
Person Profile Table
Student Info
SWRTEST
SAT Test Table
SWVCRSE
crn
Course Validation Table
SWRREGS
term_code
Course/Grade Registration Table
SWVTERM
term_code
Term Validation Table
TWRACCD
detc_code
Account Table
TWVDETC
Detail Code Validation Table
SWRSTDN
stdn_code
SWVSTDN
Student Standing Table
Student Standing Validation Table

Section P: Appendix

Lesson: Banner APIs

Jump to TOC

Description

Banner APIs are written in PL/SQL and stored in the BANINST1 schema. Each table will eventually have an API associated with it (except rule and validation tables). This is only a high-level overview of APIs. For more information on APIs consult the API Developer's Guide or consider taking the API Course.

PL/SQL Packages

There are two parts to a table level API – DML and Validation. Each DML API consists of a sing stored package while the Validation API will be three or more packages.

The DML portion of the API is easily identified using the naming convention DML_<tablename>. This package contains the replacements for standard INSERT, UPDATE, and DELETE statements against that table. Instead of issuing these standard standard statements against the table, the API is called instead. The DML API is never called directly – you need to call the Validation Package which will validate the data before calling the DML package internally.

The package for the Validation portion of the API is more difficult to locate as it bears the English name of the API. Each Validation API begins with the letter representing the module (G – General, P – Payroll, S – Student, F – Finance, etc) and B as the second letter for Business Entity API. For example, the GOREMAL table validation API name is GB_EMAIL. There will also be a minimum of two other associated packages GB_EMAIL_STRINGS and GB_EMAIL_RULES. The STRINGS package contains error messages that are returned by the API. The RULES package contains the business rules or validation applied to the data for the table. There may be additional packages for user defined rules that are called by the API (called User Exits).

Section P: Appendix

Lesson: Calling APIs

Jump to TOC

Calling an API

Each high level Validation API (e.g. GB_EMAIL without the STRINGS or RULES extension) will contain the following procedures:

P_CREATE – for inserting records into the table
P_UPDATE – for updating existing records
P_DELETE – for removing records from the table

Instead of issuing an INSERT statement against a table with an API you should call P_CREATE, passing as parameters the values for each column in the table (optional columns may be omitted).

When calling APIs to manipulate table records, you can no longer use the standard COMMIT or ROLLBACK statements. You must call GB_COMMON.P_COMMIT or GB_COMMON.P_ROLLBACK for those functions. These special procedures are designed to properly process API messages from validation problems that may have occurred.

Example (from BWGKOADR Self Service package):

```
   DECLARE
      lv_rowid gb_common.internal_record_id_type;
      lv_seqno PLS_INTEGER;
   BEGIN
      gb_telephone.p_create (
                             p_pidm => pidm,
                             p_tele_code => teles (i),
                             p_phone_area => areas (i),
                             p_phone_number => phones (i),
                             p_phone_ext => exts (i),
                             p_atyp_code => atyp,
                             p_addr_seqno => aseqno,
                             p_primary_ind => NULL,
                             p_unlist_ind => unl_tab (i),
                             p_intl_access => accss (i),
                             p_data_origin => gb_common.DATA_ORIGIN,
                             p_user_id => gb_common.f_sct_user,
                             p_seqno_out => lv_seqno,
                             p_rowid_out => lv_rowid
                            );
      gb_common.p_commit;
   END;
/
```

Section Q: Self Check - Answer Key

Lesson: Section B

Jump to TOC

Exercise 1

What are the three main parts of a PL/SQL block?

Declaration
Execution
Exception

Also acceptable are: Header (for stored PL/SQL) Declare, Begin/End, and Exception

Exercise 2

Describe an anonymous block.

A block that has no name and is not stored in the database.

Section Q: Self Check - Answer Key

Lesson: Section C

Jump to TOC

Exercise 1

How can you create a variable that has the same characteristics as a column in the database?

Use the %TYPE designator (e.g. my_var table.column%TYPE)

Exercise 2

Mark the following as legal or illegal definitions:

```
lv_Pidm  NUMBER;                    (Legal)
lv_Pidm  NUMBER(8) :=  null;        (Legal)
lv_PIDM  NUMBER(8) := 0;            (Legal)

lv_PIDM  NUMBER(8) NOT NULL;        (Illegal)
    Must supply a value when using NOT NULL

lv_name  varchar2;                  (Illegal)
    Must supply a size when declaring VARCHAR2 datatypes

lv_state char;                      (Legal)
    NOTE:  while legal creates a character variable with length one (char(1))

lv_today date := '01/01/2003';      (Illegal)
    If not in Oracle's default date format must use to_date function with format mask

lv_state varchar2(4) := 'FLORIDA';  (Illegal)
    Assignment is larger than variable declaration

update   varchar2(7) := 'FLORIDA';  (Illegal)
    Cannot use reserved words to declare variables
```

Section Q: Self Check - Answer Key

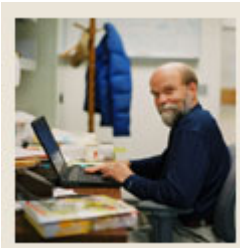
Lesson: Section D

Jump to TOC

Exercise 1

Write a PL/SQL block to insert a row about you into SWRIDEN. The values for the insert must be declared in a declaration section. (Either prompt for the values using the *&variable* convention or hard code the values in the declaration section.) Check to make sure your row was inserted.

```
DECLARE
   lv_pidm          NUMBER (8)    := &pidm;
   lv_last_name     VARCHAR2 (30) := '&last_name';
   lv_first_name    VARCHAR2 (15) := '&first_name';
   lv_id            VARCHAR2 (9)  := '&id';
BEGIN
   INSERT INTO swriden
        (swriden_pidm, swriden_ID, swriden_last_name,
         swriden_first_name, swriden_activity_date)
   VALUES (lv_pidm, lv_id, lv_last_name,
           lv_first_name, SYSDATE);
  COMMIT;
END;
/
  Enter value for pidm: 87654
  Enter value for last_name: Scrooge
  Enter value for first_name: Ebenezer
  Enter value for id: 333445555

  PL/SQL procedure successfully completed.

  SQL> select * from swriden where swriden_pidm = 87654;

  SWRIDEN_PIDM SWRIDEN_I SWRIDEN_LAST_NAME         SWRIDEN_FIRST_N
SWRIDEN_MI
   ------------ --------- ------------------------ -------------- --
-------------

   S SWRIDEN_A
   - ---------
         87654 333445555 Scrooge                  Ebenezer
     22-DEC-06


   1 row selected.
```



Section Q: Self Check - Answer Key

Lesson: Section E

Jump to TOC

Exercise 1

What are the four types of loops?

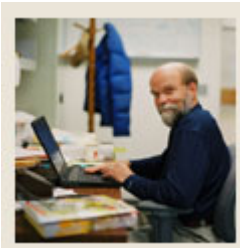
- **Simple Loops**
- **Numeric Loops**
- **WHILE Loops**
- **Cursor FOR Loops**

Exercise 2

Write a PL/SQL block that will conditionally execute for the following conditions. Your script should ask the user for a value.

- If x = 20, then add 10
- If x = 30, then add 20
- If x = 40, then add 30
- If x = 50, then add 40

```
declare
  lv_in     number(2);
  lv_out    number(2);
  lv_error  varchar2(50) := 'Enter a number 20 to 50; multiples of 10';
begin
  lv_in := &enter_number;
  case lv_in
    when 20 then lv_out := lv_in + 10;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    when 30 then lv_out := lv_in + 20;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    when 40 then lv_out := lv_in + 30;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    when 50 then lv_out := lv_in + 40;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    else
      dbms_output.put_line(lv_error);
   end case;
end;
/
```



Section Q: Self Check - Answer Key

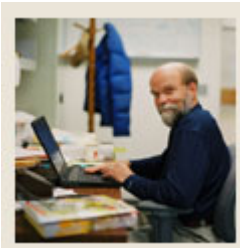
Lesson: Section E (Continued)

Jump to TOC

Exercise 3

Using a loop, write a PL/SQL block that inserts values and the date that value was calculated into the TEMP table. Values should be between 1 and 20. Check the TEMP table to make sure the values were added.

```
BEGIN
  FOR my_index IN 1..20 LOOP
     INSERT INTO temp (col1, col2, Message)
            VALUES (my_index, sysdate, 'Exercise E-4');
  END LOOP;
  COMMIT;
END;
/

  SQL> Select * from temp;
```



Section Q: Self Check - Answer Key

Lesson: Section F

Jump to TOC

Exercise 1

What are three advantages of using PL/SQL Error Handling?

Some examples:

- **Event driven handling of errors**
- **Separation of error-processing code**
- **No need to code multiple checks**
- **Isolates error-handling routines and improves readability of code**

Exercise 2

What are three types of PL/SQL exceptions?

- **Named system exceptions**
- **Named programmer-defined exceptions**
- **Unnamed system exceptions**

Exercise 3

Identify the type of PL/SQL exceptions in the following examples:

```
… DECLARE my_exception  Exception;
```

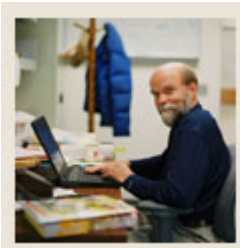
Named programmer-defined exceptions

```
…PRAGMA EXCEPTION_INIT my_exception, 1025;
```

Unnamed system exception

```
DECLARE…
BEGIN…
…….
EXCEPTION
WHEN OTHERS THEN
    ROLLBACK;
END;
end
```

Unnamed system exception



Section Q: Self Check - Answer Key

Lesson: Section F (Continued)

Jump to TOC

Exercise 4

Redo Exercise 1 from Section E to raise a user-defined exception if the number entered is not 20, 30, 40, or 50. Process the error in an exception handler and display an appropriate message.

```
DECLARE
  lv_in     number(2);
  lv_out    number(2);
  lv_error  varchar2(50) := 'Enter a number 20 to 50; multiples of 10';
  invalid_entry   EXCEPTION;
BEGIN
  lv_in := &enter_number;
  case lv_in
    when 20 then lv_out := lv_in + 10;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    when 30 then lv_out := lv_in + 20;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    when 40 then lv_out := lv_in + 30;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    when 50 then lv_out := lv_in + 40;
      dbms_output.put_line('Old: ' || lv_in || ' New: ' || lv_out);
    else
      raise invalid_entry;
   end case;

  EXCEPTION
    WHEN invalid_entry THEN
        dbms_output.put_line(lv_error);
END;
/
```

Section Q: Self Check - Answer Key

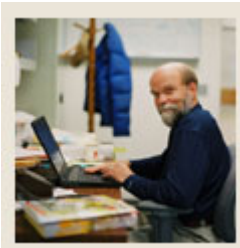
Lesson: Section G

Jump to TOC

Exercise 1

Write a PL/SQL script using a cursor to display the id, last_name and first_name from SWRIDEN.

```
  DECLARE
    cursor name_cursor is
    SELECT swriden_id, swriden_last_name, swriden_first_name
      FROM swriden
     WHERE swriden_change_ind is null;
     lv_id swriden.swriden_id%type;
     lv_lname swriden.swriden_last_name%type;
     lv_fname swriden.swriden_first_name%type;

  BEGIN
    dbms_output.enable;
    OPEN name_cursor;
    LOOP
      FETCH name_cursor into lv_id, lv_lname, lv_fname;
      EXIT WHEN name_cursor%notfound;
      dbms_output.put_line(lv_id||' '|| lv_lname||' '|| lv_fname);
    END loop;
    CLOSE name_cursor;
  END;
/
```



Section Q: Self Check - Answer Key

Lesson: Section G (Continued)

Jump to TOC

Exercise 2

Write a PL/SQL script that prompts for a pidm, and selects all columns from the person table, SWBPERS, based upon that pidm. Rather than explicitly declaring all the host variables, use %ROWTYPE. Select the columns by using the SELECT INTO statement. Display the SSN and birth date variables using DBMS_OUTPUT.PUT_LINE. If no record is found, then display an error message to the user.

```
  DECLARE
     swbpers_rec swbpers%ROWTYPE;
  BEGIN
     DBMS_OUTPUT.ENABLE(20000);
     SELECT *
       INTO swbpers_rec
       FROM swbpers
      WHERE swbpers_pidm = &PIDM;
     DBMS_OUTPUT.PUT_LINE(swbpers_rec.swbpers_ssn||' '||
          swbpers_rec.swbpers_birth_date);
  EXCEPTION
     WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No person record for this pidm ');
  END;
/
```

Section Q: Self Check - Answer Key

Lesson: Section G (Continued)

Jump to TOC

Exercise 3

Create a PL/SQL script which selects the pidm, id, first_name || last_name (column alias of 'name') and change indicator from SWRIDEN. Select both current rows (change_ind is null) and non-current rows from SWRIDEN (change_ind is NOT null). Sort by pidm and change indicator.

Write each pidm, id, and name using the DBMS_OUTPUT package. If the change indicator is null, specify that the record is 'Current' when writing the line. If the change indicator is not null, specify 'Historical'. Run your script.

```
  DECLARE
     CURSOR swriden_cursor IS
        SELECT swriden_pidm, swriden_id, swriden_first_name
               ||' '||swriden_last_name name,
               DECODE(swriden_change_ind, null, 'Current',
                      'Historical') historical_ind
          FROM swriden
      ORDER BY swriden_pidm, swriden_activity_date;
  BEGIN
     DBMS_OUTPUT.ENABLE(20000);
     FOR swriden_rec IN swriden_cursor LOOP
     DBMS_OUTPUT.PUT_LINE(swriden_rec.swriden_pidm||'   '||
        swriden_rec.swriden_id||'   '||swriden_rec.name||'   '||
        swriden_rec.historical_ind);
     END LOOP;
  END;
/
```

Exercise 4

Starting with Exercise 3, using an IF statement, alter your script so that the person information (SSN and birth date) is displayed for current rows and is not displayed for the historical records (change indicator is not null).

```
DECLARE
   CURSOR swriden_cursor IS
      SELECT swriden_pidm, swriden_id, swriden_first_name
             ||' '||swriden_last_name name,
             DECODE(swriden_change_ind,null, 'Current',
                    'Historical') historical_ind
        FROM swriden
       ORDER BY swriden_pidm, swriden_activity_date;
   lv_birth_date swbpers.swbpers_birth_date%TYPE;
   lv_ssn        swbpers.swbpers_ssn%TYPE;
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   FOR swriden_rec IN swriden_cursor LOOP
     DBMS_OUTPUT.PUT_LINE(swriden_rec.swriden_pidm||'   '||
        swriden_rec.swriden_id||'   '||swriden_rec.name||'   '||
        swriden_rec.historical_ind);
     IF swriden_rec.historical_ind = 'Current' THEN
        BEGIN
           SELECT swbpers_birth_date,swbpers_ssn
             INTO lv_birth_date,lv_ssn
             FROM swbpers
            WHERE swbpers_pidm = swriden_rec.swriden_pidm;
           DBMS_OUTPUT.PUT_LINE(lv_birth_date||' '||lv_ssn);
        EXCEPTION
           WHEN NO_DATA_FOUND THEN
              DBMS_OUTPUT.PUT_LINE('No SWPBERS record for this
person.');
        END;
     END IF;
   END LOOP;
END;
/
```

Exercise 5

Create a PL/SQL script that selects the pidm and birth date from SWBPERS into a PL/SQL table. Sort by birth date.

Using the PL/SQL table, display the pidm and birth date when the birth date is equal to or greater than '01-JAN-1970', evaluating each record one at a time. Once a record's information is displayed, remove the record from the PL/SQL Table (**NOT the SWBPERS table**).

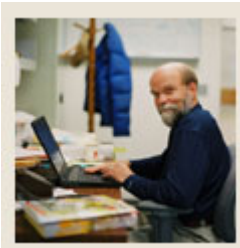
```
  DECLARE
     TYPE persontabtype IS TABLE OF swbpers%ROWTYPE
        INDEX BY BINARY_INTEGER;
     person_rec persontabtype;
     lv_counter    NUMBER(6) := 0;
     CURSOR person_cursor IS
        SELECT *
          FROM swbpers
         ORDER BY swbpers_birth_date;
  BEGIN
     DBMS_OUTPUT.ENABLE(20000);
     OPEN person_cursor;
     LOOP
        lv_counter := lv_counter + 1;
        FETCH person_cursor INTO person_rec(lv_counter);
        EXIT WHEN person_cursor%NOTFOUND;
        IF person_rec(lv_counter).swbpers_birth_date >=
              TO_DATE('01-JAN-1970','DD-MON-YYYY') THEN
           DBMS_OUTPUT.PUT_LINE(person_rec(lv_counter).swbpers_pidm||
              ' '||person_rec(lv_counter).swbpers_birth_date);
           person_rec.delete(lv_counter);
        END IF;
     END LOOP;
     CLOSE person_cursor;
  END;
/
```

Exercise 6

Using the script from the previous exercise, display the pidm and birth date for all records where the birth date is less than '01-JAN-1970'. You should not have to reevaluate the birth date condition, because records whose birth dates were equal to or greater than '01-JAN-1970' were deleted in the previous step.

Note: Be sure to make use of some PL/SQL table attributes.

```
DECLARE
   TYPE persontabtype IS TABLE OF swbpers%ROWTYPE
      INDEX BY BINARY_INTEGER;
   person_rec       persontabtype;
   lv_counter       NUMBER(6) := 0;
   lv_total_records NUMBER(6) := 0;
   CURSOR person_cursor IS
      SELECT *
        FROM swbpers
       ORDER BY swbpers_birth_date;
   lv_date VARCHAR2(11) := '01-JAN-1970';
BEGIN
   DBMS_OUTPUT.ENABLE(20000);
   OPEN person_cursor;
   LOOP
      lv_counter := lv_counter + 1;
      FETCH person_cursor INTO person_rec(lv_counter);
      EXIT WHEN person_cursor%NOTFOUND;
      IF person_rec(lv_counter).swbpers_birth_date >=
            TO_DATE(lv_date,'DD-MON-YYYY') THEN
--         DBMS_OUTPUT.PUT_LINE(person_rec(lv_counter).swbpers_pidm||
--            ' '||person_rec(lv_counter).swbpers_birth_date);
         person_rec.delete(lv_counter);
      END IF;
   END LOOP;
   CLOSE person_cursor;
   lv_total_records := person_rec.count;
   lv_counter := person_rec.first;
   DBMS_OUTPUT.PUT_LINE(chr(10)||'Records Before '||lv_date);
   FOR i IN 1..lv_total_records LOOP
      DBMS_OUTPUT.PUT_LINE(person_rec(lv_counter).swbpers_pidm
          ||' '||person_rec(lv_counter).swbpers_birth_date);
      lv_counter := person_rec.next(lv_counter);
   END LOOP;
END;
```



Section Q: Self Check - Answer Key

Lesson: Section H

Jump to TOC

Exercise 1

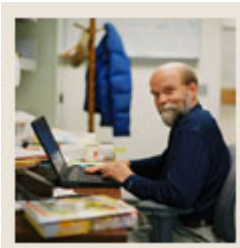
In the account table TWRACCD, an amount is considered unpaid if the PAID_DATE is null. Create a stored function called calc_amt_owed, which returns the sum amount of unpaid bills when a pidm is passed.

```
CREATE OR REPLACE FUNCTION calc_amt_owed (pi_pidm in NUMBER) RETURN
NUMBER
IS
   lv_total_owed NUMBER(9,2) := 0;
BEGIN
   SELECT SUM(twraccd_amount)
     INTO lv_total_owed
     FROM twraccd
    WHERE twraccd_PAID_DATE IS NULL
      AND twraccd_pidm = pi_pidm;
   RETURN lv_total_owed;
EXCEPTION
   WHEN NO_DATA_FOUND THEN RETURN 0;
END;
/
```

Exercise 2

Select the first name, last name, and amount owed, using the function you just created.

```
SELECT swriden_first_name, swriden_last_name,
calc_amt_owed(swriden_pidm)
  FROM swriden
 WHERE swriden_change_ind is null;
```



Section Q: Self Check - Answer Key

Lesson: Section H (Continued)

Jump to TOC

Exercise 3

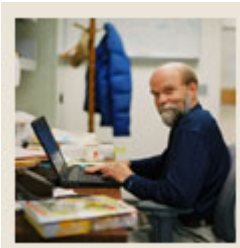
Create a procedure called insert_acct_info that inserts an account transaction into the account table (TWRACCD). The procedure should have parameters for:

- PIDM
- TERM_CODE
- DETC_CODE
- TRANS_TYPE
- BILL_DATE
- PAID_DATE
- AMOUNT

ACTIVITY_DATE should be automatically derived within the procedure.

Be sure to test your procedure by calling the procedure to insert a record.

```
CREATE OR REPLACE PROCEDURE insert_acct_info
    (pi_pidm IN NUMBER, pi_term_code IN VARCHAR2,
     pi_detc_code IN VARCHAR2, pi_trans_type IN VARCHAR2,
     pi_bill_date IN DATE, pi_paid_date IN DATE, pi_amount IN NUMBER)
IS
BEGIN
   DBMS_OUTPUT.ENABLE;
   INSERT INTO twraccd (twraccd_pidm, twraccd_term_code,
          twraccd_detc_code, twraccd_trans_type, twraccd_bill_date,
          twraccd_paid_date, twraccd_amount, twraccd_activity_date)
     VALUES (pi_pidm, pi_term_code, pi_detc_code, pi_trans_type,
             pi_bill_date, pi_paid_date, pi_amount, SYSDATE);
     COMMIT;
EXCEPTION
   WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE('Error inserting into twraccd.');
END;
/


BEGIN
     insert_acct_info(&pidm, &term_code, '&detc_code', '&trans_type',
            '&bill_date', '&paid_date', &amount);
END;
/
```



Section Q: Self Check - Answer Key

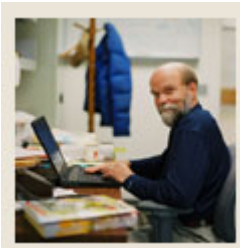
Lesson: Section H (Continued)

Jump to TOC

Exercise 4

Create a procedure that inserts into the TEMP table when an error occurs in a script (**replacing the bolded code below**). The procedure should pass in the error code and error message.

```
  ...
WHEN OTHERS THEN
           lv_sqlcode := SQLCODE;
           lv_sqlerrm := SUBSTR(SQLERRM, 1, 55);
           ROLLBACK;
           INSERT INTO temp (col1, col2, message)
           VALUES (lv_sqlcode, sysdate, lv_sqlerrm);
           COMMIT;


  CREATE OR REPLACE PROCEDURE insert_errors
       (pi_err_code IN NUMBER, pi_err_msg IN VARCHAR2) IS
  BEGIN
     DBMS_OUTPUT.ENABLE;
     INSERT INTO TEMP (col1, col2, message)
        VALUES (pi_err_code, sysdate, pi_err_msg);
    EXCEPTION
     WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('Error inserting into TEMP.');
  END;
/
```



Section Q: Self Check - Answer Key

Lesson: Section H (Continued)

Jump to TOC

Exercise 5

Write a quick PL/SQL script that causes an error, such as retrieving multiple rows into a SELECT INTO statement, and calls your procedure in Exercise 4 when it encounters the error. Select from the TEMP table to make sure that the error handler is working properly.

```
  DECLARE
     lv_dummy        VARCHAR2(1);
     lv_sqlcode      NUMBER(6);
     lv_sqlerrm      VARCHAR2(55);
  BEGIN
     SELECT swriden_last_name
       INTO lv_dummy
       FROM swriden;
  EXCEPTION
     WHEN OTHERS THEN
        lv_sqlcode := SQLCODE;
        lv_sqlerrm := SUBSTR(SQLERRM, 1,55);
        Insert_errors(lv_sqlcode, lv_sqlerrm);
  END;
/
  SQL> Select * from temp;
```

Exercise 6

Locate your newly created PL/SQL objects in the database using USER_OBJECTS, USER_SOURCE, and USER_DEPENDENCIES.

```
  SQL> SELECT object_name, status FROM USER_OBJECTS
   2    WHERE object_type in ('FUNCTION','PROCEDURE');

  SQL> SELECT line, text FROM user_source;

  SQL> SELECT name, type, referenced_owner, referenced_name
   2     FROM user_dependencies
```

Section Q: Self Check - Answer Key

Lesson: Section I

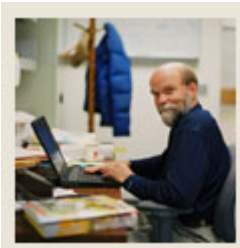
Jump to TOC

Exercise 1

e. Create a package called 'Account' containing both the insert_acct_info procedure and the calc_amt_owed function created in Section H.
f. Overload the account package, so that there are now two functions are called calc_amt_owed. The second function should accept the parameters of pidm and term code, and return the sum of the amount owed for that term code.
g. Use what you have learned to add to the exception handler for the insert_acct_info procedure, so that the message displayed indicates the nature of the error (i.e. incorrect parameters).
h. Execute the procedure and both functions from the package.

```
CREATE OR REPLACE PACKAGE ACCOUNT IS
   FUNCTION calc_amt_owed (pi_pidm IN NUMBER) RETURN NUMBER;

   FUNCTION calc_amt_owed (pi_pidm IN NUMBER,
                           pi_term IN VARCHAR2) RETURN NUMBER;

   PROCEDURE insert_acct_info (pi_pidm IN NUMBER,
         pi_term_code IN VARCHAR2, pi_detc_code IN VARCHAR2,
         pi_trans_type IN VARCHAR2, pi_bill_date IN DATE,
         pi_paid_date IN DATE, pi_amount IN NUMBER);
END ACCOUNT;
/
CREATE OR REPLACE PACKAGE BODY ACCOUNT
IS
   FUNCTION calc_amt_owed (pi_pidm in NUMBER) RETURN NUMBER
   IS
      lv_total_owed NUMBER(9,2) := 0;
   BEGIN
      SELECT NVL(SUM(twraccd_amount),0)
        INTO lv_total_owed
        FROM twraccd
       WHERE twraccd_PAID_DATE IS NULL
         AND twraccd_pidm = pi_pidm;
      RETURN lv_total_owed;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN RETURN 0;
   END calc_amt_owed;
-----------------------------------------------------------------
```

(Continued…)



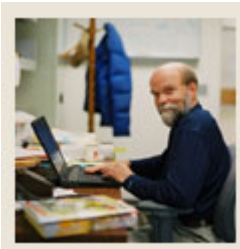
Section Q: Self Check - Answer Key

Lesson: Section I (Continued)

Jump to TOC

Exercise 1 (cont.)

```
  FUNCTION calc_amt_owed (pi_pidm IN NUMBER, pi_term IN VARCHAR2)
        RETURN NUMBER
     IS
        lv_total_owed   NUMBER (9, 2) := 0;
    BEGIN
       SELECT NVL(SUM(twraccd_amount),0)
         INTO lv_total_owed
         FROM twraccd
        WHERE twraccd_paid_date IS NULL
          AND twraccd_pidm = pi_pidm AND twraccd_term_code = pi_term;

      RETURN lv_total_owed;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
           RETURN 0;
END calc_amt_owed;
-------------------------------------------------------------------
PROCEDURE insert_acct_info
    (pi_pidm IN NUMBER, pi_term_code IN VARCHAR2,
     pi_detc_code IN VARCHAR2, pi_trans_type IN VARCHAR2,
     pi_bill_date IN DATE, pi_paid_date IN DATE, pi_amount IN NUMBER)
IS
   BEGIN
     DBMS_OUTPUT.ENABLE;
     INSERT INTO twraccd (twraccd_pidm, twraccd_term_code,
            twraccd_detc_code, twraccd_trans_type, twraccd_bill_date,
            twraccd_paid_date, twraccd_amount, twraccd_activity_date)
     VALUES (pi_pidm, pi_term_code, pi_detc_code, pi_trans_type,
             pi_bill_date, pi_paid_date, pi_amount, SYSDATE);
     COMMIT;
   EXCEPTION
     WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('Error inserting into twraccd.'
         ||SUBSTR(SQLERRM, 1,120));
   END insert_acct_info;

END ACCOUNT;
/
```



Section Q: Self Check - Answer Key

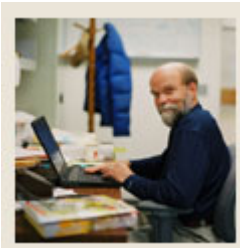
Lesson: Section I (Continued)

Jump to TOC

Then execute the following to test the functions and the procedure:

```
-- test the smart function:

SELECT SWRIDEN_FIRST_NAME, SWRIDEN_LAST_NAME,
TO_CHAR(account.calc_amt_owed(swriden_pidm),'$99,990.00') owed
  FROM swriden
 WHERE SWRIDEN_CHANGE_IND IS NULL;

SELECT SWRIDEN_FIRST_NAME, SWRIDEN_LAST_NAME,
TO_CHAR(account.calc_amt_owed(swriden_pidm,'200701'),'$99,990.00') owed
  FROM swriden
 WHERE SWRIDEN_CHANGE_IND IS NULL;
```

```
-- test the procedure
BEGIN
     account.insert_acct_info
     (12340,'200701','BOOK','C','01-SEP-2007','15-SEP-2007',456.10);
END;
/

-- force an error (term code too long) to test the exception handler
BEGIN
   account.insert_acct_info
   (12340,'2007011','TUIT','C','01-SEP-2007',null,1200);
END;
/
```



Section Q: Self Check - Answer Key

Lesson: Section J

Jump to TOC

Exercise 1

Write a script that generates a random number and a random string. Use the package DBMS_RANDOM to generate the values and DBMS_OUTPUT to display them.

```
DECLARE
      lv_random_string       VARCHAR2(50);
      lv_random_number       NUMBER;
BEGIN
      SELECT DBMS_RANDOM.NORMAL
        INTO lv_random_number
        FROM dual;
      DBMS_OUTPUT.put_line('Random Number: ' || lv_random_number);
      SELECT DBMS_RANDOM.STRING('U',22)
        INTO lv_random_string
        FROM dual;
      DBMS_OUTPUT.put_line('Random String: ' || lv_random_string);
 END;
/
```

(Your results may vary)

```
Random Number: .22365196231991325451288835283992870991 82
Random String: NCYGBBUCLCWXYYNFQBXHJX

PL/SQL procedure successfully completed.
```

Section Q: Self Check - Answer Key

Lesson: Section J (continued)

Jump to TOC

Exercise 2

Write a script to use the SYS_CONTEXT built in and at least 3 of the attributes to display information about your session. Display those values using DBMS_OUTPUT.

```
DECLARE
    lv_userenv1         VARCHAR2(50) := 'SESSION_USER';
    lv_userenv2         VARCHAR2(50) := 'HOST';
    lv_userenv3         VARCHAR2(50) := 'NLS_DATE_FORMAT';
    lv_userenv_val1     VARCHAR2(50);
    lv_userenv_val2     VARCHAR2(50);
    lv_userenv_val3     VARCHAR2(50);
BEGIN
    SELECT SYS_CONTEXT ('USERENV', lv_userenv1)
      INTO lv_userenv_val1
      FROM dual;
    DBMS_OUTPUT.put_line(lv_userenv1 || ': ' || lv_userenv_val1);
    SELECT SYS_CONTEXT ('USERENV', lv_userenv2)
      INTO lv_userenv_val2
      FROM dual;
    DBMS_OUTPUT.put_line(lv_userenv2 || ': ' || lv_userenv_val2);
    SELECT SYS_CONTEXT ('USERENV', lv_userenv3)
      INTO lv_userenv_val3
      FROM dual;
    DBMS_OUTPUT.put_line(lv_userenv3 || ': ' || lv_userenv_val3);
END;
/

SESSION_USER: TRAIN01
HOST: SCT\MAL12345
NLS_DATE_FORMAT: DD-MON-RR

PL/SQL procedure successfully completed.
```

Section Q: Self Check - Answer Key

Lesson: Section J (continued)

Jump to TOC

Exercise 3

Create a table with a CLOB column and an Identifier column. Insert a row into the table using the EMPTY_CLOB() built in. Write a string to the CLOB record you just created. Append the same value to that CLOB. Show the length of the LOB after the write and append.

Run your procedure again. What happens to the length and why?

```
create table lob_table (
    lob_id    number(3),
    document   CLOB);

insert into lob_table values(3,empty_clob());

commit;


DECLARE
     lv_lobholder    CLOB;
     lv_buffer       VARCHAR2(32000);
     lv_oldsize      NUMBER := 20;
     lv_newsize      NUMBER;
     lv_offset       NUMBER := 1;
BEGIN
     --Initialize buffer with data to be inserted
     lv_buffer := 'abc123def456ghi789jkl012mno345pqr678stu910vwx234yz56';
     lv_oldsize := length(lv_buffer);
     dbms_output.put_line(lv_buffer);
     dbms_output.put_line(to_char(lv_oldsize));
     SELECT document INTO lv_lobholder  -- get LOB handle
       FROM lob_table
       WHERE lob_id = 3 FOR UPDATE;
     dbms_lob.write(lv_lobholder,lv_oldsize,lv_offset,lv_buffer);
     dbms_lob.append(lv_lobholder,lv_buffer);
     lv_newsize := dbms_lob.getlength(lv_lobholder);
     dbms_output.put_line(lv_newsize);
     COMMIT;
END;
/
```

Section Q: Self Check - Answer Key

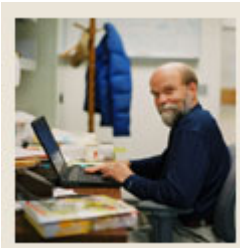
Lesson: Section J (continued)

Jump to TOC

Exercise 3 (continued)

```
abcdefghijklmnopqrstuvwxyz
52
104

PL/SQL procedure successfully completed.

SQL> /
abcdefghijklmnopqrstuvwxyz
52
156

PL/SQL procedure successfully completed.
```

The size keeps increasing because the WRITE only overwrites the first 52 characters. The APPEND keeps appending to the end. Use the ERASE command if you want to overwrite the entire contents of the LOB.



Section Q: Self Check - Answer Key

Lesson: Section K

Jump to TOC

Exercise 1

Create a sequence.

```
CREATE SEQUENCE my_seq start with 1;
```

Exercise 2

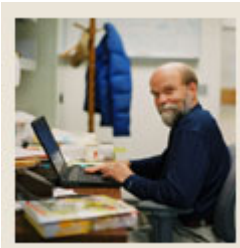
Test the sequence by selecting the first value.

```
SELECT my_seq.nextval from dual;
```

Exercise 3

Create a database trigger on SWBPERS. For each update statement, insert into the temp table a value from the above sequence, the current date, and the user making the update. Write a statement to update a row in the SWBPERS table and view the results of the trigger in the temp table (don't forget to commit your update).

```
CREATE OR REPLACE TRIGGER after_swbpers
  AFTER UPDATE ON swbpers
BEGIN
   INSERT INTO temp (col1, col2, message)
      VALUES (my_seq.nextval, SYSDATE, user);
END After_swbpers;

UPDATE swbpers SET SWBPERS_SSN = 111223333
WHERE swbpers_pidm = 12345;
COMMIT;

SELECT * from temp where col2 like sysdate;


      COL1 COL2      MESSAGE
---------- --------- ------------------------------
         1 24-NOV-06 TRAIN01
```



Section Q: Self Check - Answer Key

Lesson: Section K (Continued)

Jump to TOC

Exercise 4

Create a database trigger on the SWRIDEN table. For every row inserted, check to see if a current row exists (change_ind is null). Update the original current row so that the change indicator is an *I* (ID change) or an *N* (name change), depending on the type of change. Write an insert statement for the SWRIDEN table that results in a change to an existing record's ID or Name (do not insert a record for a new person). After committing your insert, check the table to see if the previous record for that PIDM was updated correctly.

```
CREATE OR REPLACE TRIGGER check_swriden_row
   BEFORE INSERT ON swriden
   FOR EACH ROW
BEGIN
DECLARE
   lv_lname swriden.swriden_last_name%type;
   lv_fname swriden.swriden_first_name%type;
   lv_id swriden.swriden_id%type;
   lv_ch_ind_val swriden.swriden_change_ind%type;
BEGIN
   SELECT swriden_id, swriden_last_name, swriden_first_name
     INTO lv_id, lv_lname, lv_fname
     FROM swriden
    WHERE swriden_change_ind IS NULL
      AND swriden_pidm = :new.swriden_pidm;

   IF lv_id != :new.swriden_id
     THEN lv_ch_ind_val := 'I';
     ELSIF (lv_lname != :new.swriden_last_name)
        OR (lv_fname != :new.swriden_first_name)
       THEN lv_ch_ind_val := 'N';
   END IF;

   UPDATE swriden
      SET swriden_change_ind = lv_ch_ind_val,
          swriden_activity_date = SYSDATE
    WHERE swriden_pidm = :new.swriden_pidm
      AND swriden_change_ind IS NULL;
END;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
           NULL;
END check_swriden_row;
```

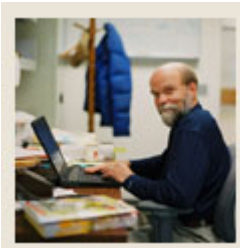
Section Q: Self Check - Answer Key

Lesson: Section K (Continued)

Jump to TOC

Exercise 4 (continued)

```
   SQL> select swriden_pidm PIDM, swriden_id ID, swriden_last_name,
               swriden_first_name, swriden_mi
          from swriden
         where swriden_change_ind is null
   SQL> /


PIDM   ID        SWRIDEN_LAST_NAME      SWRIDEN_FIRST_N SWRIDEN_MI
------ --------- ---------------------- --------------- --------------
 12340 157834585 Brown                  Julie           K
 12341 5829934   Smith                  Robert          E
 12342 3539543   Johnson                Peter           S
 12343 145672112 Jones-Erickson         Sandy           J
 12344 692568211 Erickson               Ralph           L
 12345 578549991 Erickson               Susan           T
 12346 543853339 White                  Nancy           Carol
 12347 543853339 Marx                   Joan            Elizabeth
 12348 543853339 Clifford               Stephanie       Geena
 12349 543853339 Serum                  Tracy           Paige
 87654 333445555 Scrooge                Ebenezer

11 rows selected.

SQL> insert into swriden values
         (12346,543853339,'Purple','Nancy','Carol',null,sysdate);

SQL> commit;

Commit complete.

SQL> select swriden_pidm PIDM, swriden_id, swriden_last_name,
  2         swriden_first_name FIRST, swriden_mi,
  3         swriden_change_ind CHG_IND, swriden_activity_date ACT_DATE
  4*   from swriden where swriden_pidm = 12346;

 PIDM SWRIDEN_I SWRIDEN_LAST_NAME FIRST       SWRIDEN_MI  C ACT_DATE
----- --------- ----------------- ----------- ----------- - ---------
12346 543853339 White             Nancy       Carol       N 22-DEC-06
12346 543853339 Purple            Nancy       Carol         22-DEC-06
```



Section Q: Self Check - Answer Key

Lesson: Section L

Jump to TOC

Exercise 1

Create a directory object in the database that points to a directory supplied by the instructor. To ensure each class user creates a unique directory name, include your initials in the name of the directory.

CREATE OR REPLACE DIRECTORY <YOUR_INIT>_PLSQLCLASS as

'<path supplied by instructor>';

Exercise 2

Create a package called MY_TOOLS containing a stored procedure called DISPLAY_SOURCE. The procedure should accept the parameters of a directory name, output file name, and subprogram name. The DISPLAY_SOURCE procedure will retrieve the source code from USER_SOURCE for the program passed is as the subprogram name and write it to a file.

Execute the packaged procedure, and passing the directory you created above, a file name such as <Your Initials>_ex2.txt (e.g. abc_ex2.txt), and the name of one of the stored programs you created in class (e.g. CALC_AMT_OWED, ACCOUNT, INSERT_ERRORS).

```
CREATE OR REPLACE PACKAGE my_tools IS
  PROCEDURE display_source (pi_directory IN VARCHAR2,
                            pi_filename IN VARCHAR2,
                            pi_subprogram IN VARCHAR2);
END;
/
```

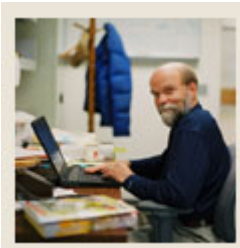
(continued…)

Section Q: Self Check - Answer Key

Lesson: Section L (Continued)

Jump to TOC

```
CREATE OR REPLACE PACKAGE BODY my_tools
IS
   PROCEDURE display_source (pi_directory IN VARCHAR2,
                             pi_filename IN VARCHAR2,
                             pi_subprogram IN VARCHAR2)
   IS
      lv_out_file   UTL_FILE.file_type;

      CURSOR source_cursor
      IS
         SELECT name, text
           FROM user_source
          WHERE name = pi_subprogram
          ORDER BY name, line;
    BEGIN
      lv_out_file := UTL_FILE.fopen (pi_directory, pi_filename, 'w');

      FOR source_rec IN source_cursor
      LOOP
         UTL_FILE.put_line (lv_out_file, source_rec.text);
      END LOOP;

      UTL_FILE.fclose(lv_out_file);

   EXCEPTION
      WHEN UTL_FILE.invalid_path
      THEN
         DBMS_OUTPUT.put_line ('Invalid path');
      WHEN OTHERS
      THEN
         DECLARE
            err_msg   VARCHAR2 (200) := SUBSTR (SQLERRM, 1, 200);
         BEGIN
            DBMS_OUTPUT.put_line (err_msg);
            IF UTL_FILE.is_open(lv_out_file) THEN
               UTL_FILE.fclose(lv_out_file);
             END IF;
         END;
   END display_source;

END my_tools;
 /
```



Section Q: Self Check - Answer Key

Lesson: Section L (Continued)

Jump to TOC

Exercise 2 (continued)

Test your procedure

```
SQL> EXECUTE MY_TOOLS.DISPLAY_SOURCE ('<DIR_NAME>', '<output file
name>', '<subprogram name>');
```

- **NOTE: parameters for DIR_NAME and SUBPROGRAM_NAME are case sensitive. Good programmers will make the parameters case insensitive by handling any case issues within the program (i.e. using and UPPER function)**
- **DIR_NAME should be the name of the directory that you assigned in Exercise 1.**
- **Output File Name should be a unique name you assign to the output file. Include your initials in the file name to distinguish it from files for the other class participants.**
- **Subprogram Name should be the name of one of the procedure or functions that you created previously (e.g. CALC_AMOUNT_OWED, INSERT_ERRORS, ACCOUNT).**

Exercise 3

Create a second stored procedure called display_source within the MY_TOOLS package (overloading the package). The procedure should only accept the parameters of a directory name and file name. It will:

- Retrieve the source of all subprograms within your schema from USER_SOURCE and write the output to a directory and file as specified by your instructor.
- Test the procedure by executing it and passing in the name of one of the procedures or functions you created previously.
- Make this version case insensitive by handling the case restrictions inside the procedure.

Additions to the package header:

```
    PROCEDURE display_source (pi_directory    IN   VARCHAR2,
                              pi_filename     IN   VARCHAR2
    );
```

Exercise 3 (continued)

Additions to package body:

```
     PROCEDURE display_source (pi_directory    IN   VARCHAR2,
                               pi_filename     IN   VARCHAR2
      )
      IS
         lv_out_file   UTL_FILE.file_type;

        CURSOR source_cursor
        IS
           SELECT text
             FROM user_source
            ORDER BY name, type, line;
     BEGIN
        lv_out_file := UTL_FILE.fopen(UPPER(pi_directory), pi_filename,
  'w');

        FOR source_rec IN source_cursor
        LOOP
           UTL_FILE.put_line (lv_out_file, source_rec.text);
        END LOOP;

        UTL_FILE.fclose(lv_out_file);
     EXCEPTION
        WHEN UTL_FILE.invalid_path
        THEN
           DBMS_OUTPUT.put_line ('Invalid path');
        WHEN OTHERS
        THEN
           DECLARE
              err_msg   VARCHAR2 (200) := SUBSTR (SQLERRM, 1, 200);
           BEGIN
              DBMS_OUTPUT.put_line (err_msg);
             IF UTL_FILE.is_open(lv_out_file) THEN
                UTL_FILE.fclose(lv_out_file);
             END IF;
           END;
     END display_source;
/
```

Exercise 3 (continued)

Testing Program:

```
SQL> EXECUTE MY_TOOLS.DISPLAY_SOURCE
        ('<DIR_NAME>','<Output File Name>');
```

Exercise 4

Create a third stored procedure within the package MY_TOOLS called LOG_ERROR. The procedure should write error messages to a log file.

- The parameters passed should be a program name and error message.
- The log file should be named the program name parameter concatenated with '.log', and in a directory created in Exercise 1.
- The procedure should write the current system time and the error message to the log file.
- Call the log_error procedure by forcing an error.

Additions to Package Header:

```
   PROCEDURE error_log (pi_directory      IN   VARCHAR2,
                        pi_program_name   IN   VARCHAR2,
                        pi_err_msg        IN   VARCHAR2
   );
```

Additions to the Package Body:

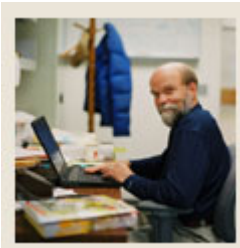
```
   PROCEDURE error_log (pi_directory      IN   VARCHAR2,
                        pi_program_name   IN   VARCHAR2,
                        pi_err_msg        IN   VARCHAR2
   )
   IS
      lv_out_file       UTL_FILE.file_type;
      lv_curr_datetime  VARCHAR2 (30)
                   := TO_CHAR (SYSDATE, 'DD-MON-YYYY:HH:MI:SS');
      lv_log_file       VARCHAR2 (30) := pi_program_name || '.log';
   BEGIN
      lv_out_file := UTL_FILE.fopen(pi_directory, lv_log_file, 'w');
      UTL_FILE.put_line (lv_out_file, CURRENT_DATE);
      UTL_FILE.put_line (lv_out_file, ' ' || pi_err_msg);
      UTL_FILE.fclose (lv_out_file);
```

Exercise 4 (continued)

```
  EXCEPTION
        WHEN UTL_FILE.invalid_operation
        THEN
           UTL_FILE.fclose_all;
           raise_application_error (-20061, 'Invalid Operation');
        WHEN UTL_FILE.invalid_filehandle
        THEN
           UTL_FILE.fclose_all;
           raise_application_error (-20062, 'Invalid File');
        WHEN UTL_FILE.write_error
        THEN
           UTL_FILE.fclose_all;
           raise_application_error (-20063, 'Write Error');
        WHEN OTHERS
        THEN
           UTL_FILE.fclose_all;
           RAISE;
     END error_log;
```

Code to test error_log procedure:

```
       DECLARE
          lv_dummy varchar2(1);
       BEGIN
          SELECT '12345' INTO LV_DUMMY
            FROM DUAL;
       EXCEPTION
          WHEN OTHERS THEN
             DECLARE
                LV_SQLERRM VARCHAR2(200) := SUBSTR(SQLERRM,1,200);
             BEGIN
                My_tools.error_log('<DIR_NAME>','myprogram',
                   lv_sqlerrm);
             END;
       END;
/
```



Section Q: Self Check - Answer Key

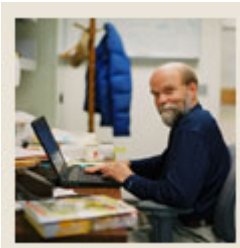
Lesson: Section M

Jump to TOC

Exercise 1

Create a public pipe named *TRAINx_PIPE*, where *x* is your training account number. Submit a message to the pipe.

```
  DECLARE
     lv_status   INTEGER;
  BEGIN
     lv_status := DBMS_PIPE.create_pipe ('TRAINx_PIPE', 3000, FALSE);

     IF lv_status = 0
     THEN
        DBMS_OUTPUT.ENABLE (20000);
        DBMS_OUTPUT.put_line ('Successfully created pipe.');
     ELSE
        DBMS_OUTPUT.put_line ('Could not create pipe.');
     END IF;
  END;
  /

  DECLARE
      lv_status       INTEGER;
  BEGIN
      DBMS_PIPE.PACK_MESSAGE('This is a test');
      lv_status := DBMS_PIPE.SEND_MESSAGE('TRAINx_PIPE');
      IF lv_status <> 0 THEN
         DBMS_OUTPUT.ENABLE(20000);
         DBMS_OUTPUT.PUT_LINE('Could not send message');
      END IF;
   END;
/
```



Section Q: Self Check - Answer Key

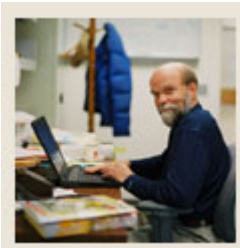
Lesson: Section M (Continued)

Jump to TOC

Exercise 2

Retrieve a message from a pipe that your neighbor submitted. Are you able to receive each other's messages? (Remember, pipe names are case sensitive.)

```
  DECLARE
     lv_message VARCHAR2(100);
     lv_status     INTEGER;
  BEGIN
     DBMS_OUTPUT.ENABLE(20000);
     status := DBMS_PIPE.RECEIVE_MESSAGE('TRAINx_PIPE');
     DBMS_PIPE.UNPACK_MESSAGE(lv_message);
     DBMS_OUTPUT.ENABLE(20000);
     DBMS_OUTPUT.PUT_LINE(lv_message);
     IF lv_status <> 0 THEN
        DBMS_OUTPUT.PUT_LINE('Could not receive message');
     END IF;
  END;
/
```



Section Q: Self Check - Answer Key

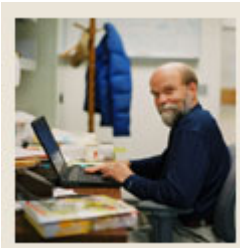
Lesson: Section M (Continued)

Jump to TOC

Exercise 3

Create an alert called *TRAINx_ALERT*, where *x* is your training account number. Work with a neighbor to make sure that each of you can register and receive notification from an alert.

```
BEGIN
    DBMS_ALERT.SIGNAL('TRAINx_ALERT',
      'The transaction has been committed.');
END;
COMMIT;


BEGIN
    DBMS_ALERT.REGISTER('TRAINx_ALERT');
END;

DECLARE
   lv_status    INTEGER;
   lv_messg     VARCHAR2 (100);
BEGIN
   DBMS_OUTPUT.ENABLE (20000);
   DBMS_ALERT.waitone ('TRAINx_ALERT', lv_messg, lv_status, 30);

   IF lv_status = 0
   THEN
      DBMS_OUTPUT.put_line (lv_messg);
   ELSIF lv_status = 1
   THEN
      DBMS_OUTPUT.put_line ('Timeout on TRAINx_ALERT.');
   END IF;
END;
/
```



Section Q: Self Check - Answer Key

Lesson: Section MN

Jump to TOC

Exercise 1

Create a procedure p_pidm_tables by running the script in your student_files folder called SectN_Ex1.sql. The procedure can be used to identify which tables and how many records per table a person has. This information may be useful in a data cleanup situation.

- The procedure accepts the parameter of pidm and schema owner.
- It retrieves tables from the view ALL_TAB_COLUMNS where the column name is like '%PIDM%'.
- Then evaluates if a row exists in each table (derived from above step) for the pidm passed in. If so, print the table name to the screen.

Execute the procedure, passing in your user ID (TRAINxx) and a PIDM from the SWRIDEN table. This will return how many records for that pidm exist in each of your tables.

```
SQL>@<path of student files>\SectN_Ex1.sql

Procedure created.

SQL> execute p_pidm_tables('TRAINxx',12345);
```

Section Q: Self Check - Answer Key

Lesson: Section MN (Continued)

Jump to TOC

Exercise 2

Write an anonymous block that uses the EXECUTE IMMEDIATE built-in to execute an SQL statement to find out how many address records a person has. Prompt for a PIDM, feeding the data to a bind variable in the SQL string. Display the results of the SQL statement execution.

```
   declare
      lv_pidm        NUMBER(8) := &pidm;
      lv_statement   VARCHAR2(200);
      lv_result      NUMBER(8);
   begin
      lv_statement := 'Select count(*) from swraddr where swraddr_pidm
= :b1';
      EXECUTE IMMEDIATE lv_statement INTO lv_result USING lv_pidm;
      DBMS_OUTPUT.put_line('PIDM ' || lv_pidm || ' has ' || lv_result
|| ' address records');
   end;
   /
```

Section Q: Self Check - Answer Key

Lesson: Section O

Jump to TOC

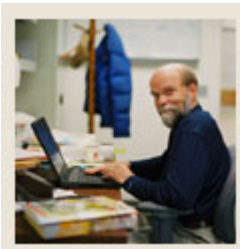
Exercise 1

Set the timing on in your SQL*Plus session and execute the following select statement (if using SQL Developer, click on the Execute Explain Plan button):

```
SELECT SWRADDR_STREET_LINE1, SWRADDR_ZIP, SWRADDR_CITY,
SWRADDR_STAT_CODE
  FROM SWRIDEN, SWRADDR
 WHERE SWRIDEN.SWRIDEN_PIDM = SWRADDR.SWRADDR_PIDM;
```

Note the amount of time it took for the statement to execute. (If using SQL Developer, the Explain Tab should have been activated when you pressed the Execute Explain Plan button.)

```
SQL> set timing on
SQL> SELECT SWRADDR_STREET_LINE1, SWRADDR_ZIP, SWRADDR_CITY,
            SWRADDR_STAT_CODE
  2    FROM SWRIDEN, SWRADDR
  3   WHERE SWRIDEN.SWRIDEN_PIDM = SWRADDR.SWRADDR_PIDM;

SWRADDR_STREET_LINE1            SWRADDR_ZI SWRADDR_CITY         SWR
------------------------------ ---------- -------------------- ---
506 BROWN STREET                19380      WEST CHESTER         PA
506 BROWN STREET                19380      WEST CHESTER         PA
210 PINE STREET                 94082      SAN FRANCISCO        CA
PO BOX 1035                     67233      BROWNVILLE           KY
23 MARKET STREET                19382      WEST CHESTER         PA
23 MARKET STREET                19382      WEST CHESTER         PA
18 CHESTNUT ROAD                23456      NEW ORLEANS          LA

7 rows selected.
```



Section Q: Self Check - Answer Key

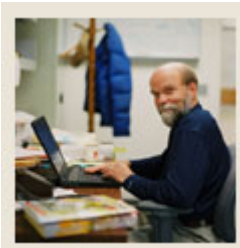
Lesson: Section O (Continued)

Jump to TOC

Exercise 1 (continued)

Explain the plan for the select statement.

```
EXPLAIN PLAN FOR
    SELECT SWRADDR_STREET_LINE1, SWRADDR_ZIP, SWRADDR_CITY,
           SWRADDR_STAT_CODE
      FROM SWRIDEN, SWRADDR
     WHERE SWRADDR.SWRADDR_PIDM = SWRIDEN.SWRIDEN_PIDM;

Explained.

SELECT RTRIM(LPAD(' ',2*LEVEL)||
            RTRIM(OPERATION)||' '||
            RTRIM(OPTIONS)||' '||
            OBJECT_NAME) QUERY_PLAN
       FROM PLAN_TABLE
CONNECT BY PRIOR ID=PARENT_ID
START WITH ID=0;

 QUERY_PLAN
 ----------------------------------

   SELECT STATEMENT
     HASH JOIN
       TABLE ACCESS FULL SWRADDR
       TABLE ACCESS FULL SWRIDEN

 Elapsed: 00:00:00.01
```



Section Q: Self Check - Answer Key

Lesson: Section O (Continued)

Jump to TOC

Exercise 2

Are indices being used, or is Oracle executing a full-table scan? If no indices are being used, then create an index for SWRADDR. Rerun Explain Plan and view the results to make sure your indices are being used. Rerun the Select statement and compare the execution time. Have you improved the performance of the statement?

```
CREATE INDEX SWRADDR_PIDM_INDEX ON SWRADDR(SWRADDR_PIDM);

DELETE PLAN_TABLE;

EXPLAIN PLAN FOR
    SELECT SWRADDR_STREET_LINE1, SWRADDR_ZIP, SWRADDR_CITY,
           SWRADDR_STAT_CODE
      FROM SWRADDR, SWRIDEN
     WHERE SWRADDR.SWRADDR_PIDM = SWRIDEN.SWRIDEN_PIDM
       AND SWRIDEN.SWRIDEN_CHANGE_IND IS NULL;
```

(Results may vary based on database version)

```
QUERY_PLAN
--------------------------------------------------

  SELECT STATEMENT
    TABLE ACCESS BY INDEX ROWID SWRADDR
      NESTED LOOPS
        TABLE ACCESS FULL SWRIDEN
        INDEX RANGE SCAN SWRADDR_PIDM_INDEX
```

Release Date

Jump to TOC

This workbook was last updated on January 25, 2008.