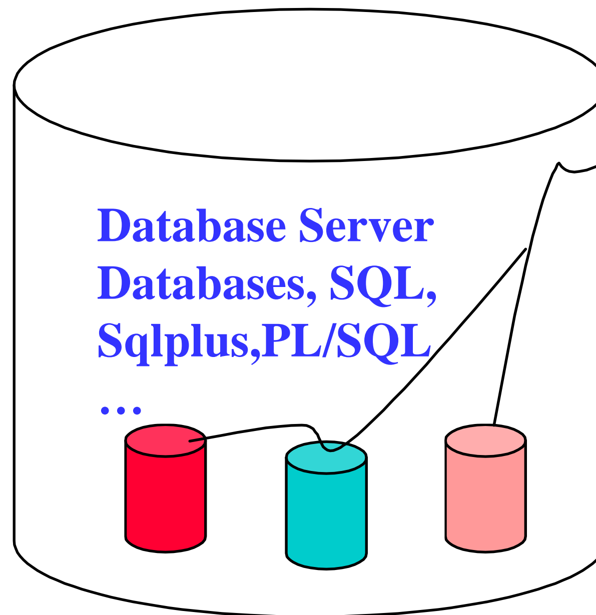
Comp-4150: Advanced and Practical Database Systems (Part B: Oracle PL/SQL)

:DB APPLICATION BUILDING

Database client Application
Query database, triggers, PL/SQL,…



Dr. C.I. Ezeife
School of Computer Science,
University of Windsor, Canada.
Email: cezeife@uwindsor.ca

Course Part B (Oracle PL/SQL) Objectives

- Broad Course Objective
- - Components of a database management system
- - Acquire database development skills necessary for building real life database applications with Oracle DBMS.
- Reference Materials
  - C.I Ezeife, *Custom Course Ware, Course Notes for Comp 4150,* Project Using Selected Tools: Advanced and Practical Database Systems (with Oracle PL/SQL and Front End Tools), University of Windsor, Fall 2021.
  - Main Course Book is Elmasri & Navathe, 7th edition, 2016
  - Benjamin Resenzweig and Elena Rakhimov, "Oracle PL/SQL by Example", Pearson, edition 5, 2015, ISBN 978-0-13-379678-0
  - Ben Forta, SQL in 10 Minutes a Day, Sams Teach Yourself, 2020, 5th edition, Print ISBN: 9780135182796, 0135182794, eText ISBN: 9780135182864, 0135182867

Course Objectives

- **Companion web site:**
  http://www.oracle.com
  (can download SQL Developer) for running PL/SQL codes. PL/SQL codes can also be run interactively with Oracle SQL*Plus.

Course Objectives

- **Part B: Oracle Database Development with Oracle PL/SQL**
  - **Oracle PL/SQL summary**
- **Part C: Database Development (GUI) (on a separate slide notes)**

Hardware and Software Requirements

- **Software Requirements/ Running Environment**
  1. Oracle DB Server (e.g., Oracle 11g or a higher version)
  2. SQL Developer
  3. Sqlplus
  4. Access to WWW
  5. Windows OS (e.g., Windows 10) and / or Unix/Linus OS

- **Hardware Requirements**
  1. A Personal Computer (e.g., 1 GHz processor, --Memory)
  2. A Unix Multiprocessor System (e.g., cs.uwindsor.ca servers)

Hardware and Software Requirements

- **Note that both the software 1 & 2 can reside on the same computer or on two separate computers. Also, while the Oracle client software [e.g., Oracle 11g client] is most suitable on a Windows based PC, the Oracle Server software can reside on a Unix machine (like CS servers Charlie/Bravo)**

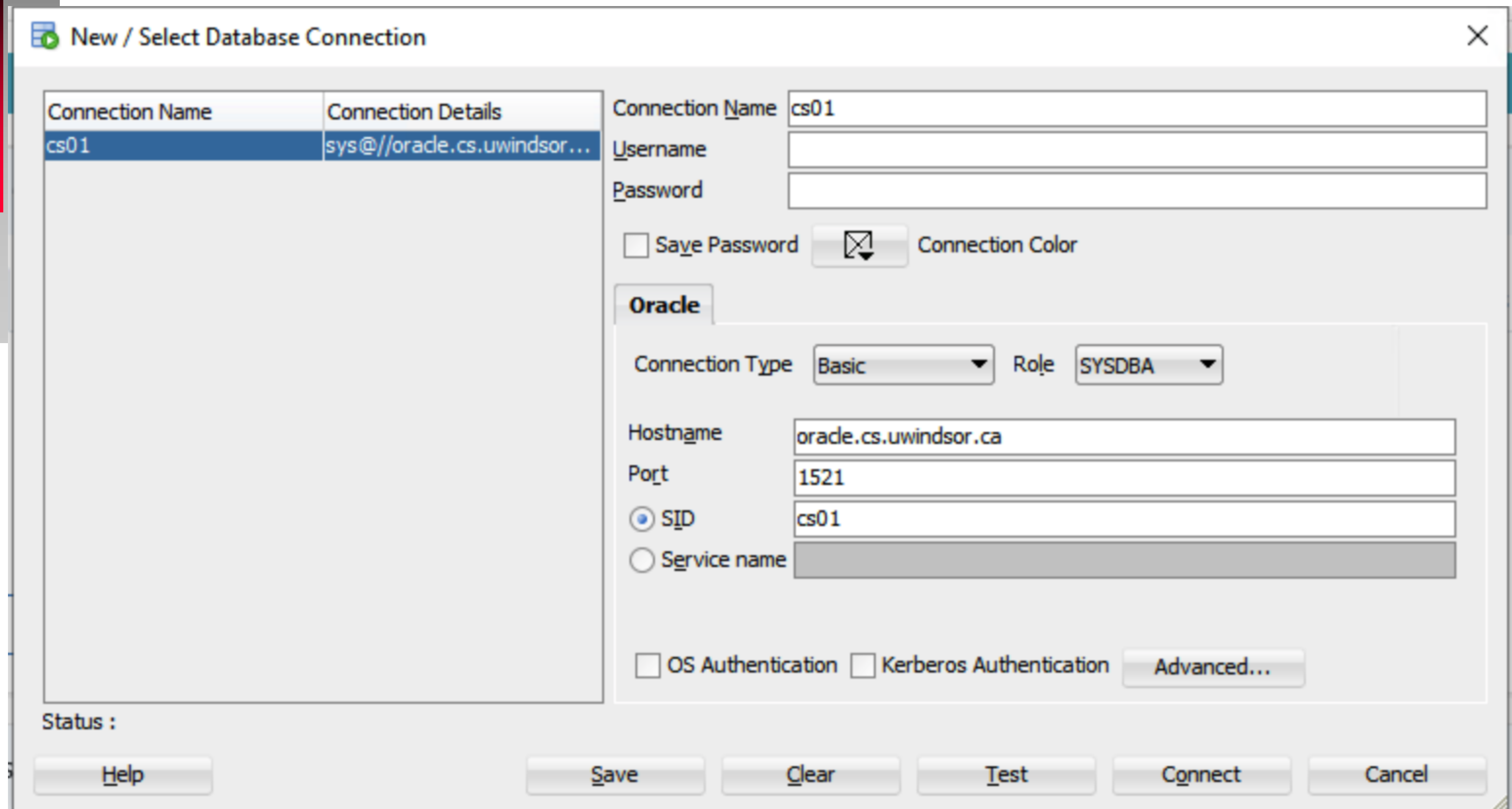
PL/SQL Development Environment : SQL Developer

- SQL Developer and SQL*Plus are two Oracle-provided tools that can be used to run PL/SQL scripts.
- SQL Developer is a free graphical tool used for database development and administration.
- It is a new addition to the Oracle tool set.
- It is a much easier tool to use than SQL*Plus as it allows you to browse database objects, run SQL statements, create, debug and run PL/SQL statements.
- Apart from the GUI format, its functionality is similar to that of the SQL*Plus discussed next.

PL/SQL Development Environment : SQL Developer

- 1. To use SQL Developer, download the tool from the Oracle website: www.oracle.com onto your desktop.
- 2. Also, for this to work on your computer at home you must install GlobalProtect VPN. Steps on how to do so can be found in the following link. (https://www.uwindsor.ca/itservices/talks/installing-globalprotect-vpn)
- 3. To use the SQL Developer to connect to your Oracle database account on our cs server, you need to establish a connection first by launching the SQL Developer and connecting as shown in the figure next with the connection strings shown
- - The username and password are your Uwindsor's that had been synchronized previously. Hostname is: oracle.cs.uwindsor.ca
- - Connection name is: CS01; SID is: CS01; Port is: 1521

PL/SQL Development Environment : SQL Developer



PL/SQL Development Environment : SQLPLUS

- Sqlplus is the software for executing SQL stmts (Sqlplus is to SQL stmts what C compiler is to C programs)

- How to end an SQL command in Sqlplus

SQL command can be ended in Sqlplus and SQL Developer in one of the following 3 ways:

  - with a semicolon (;)
  - with a forward (/) on a line
  - with a blank line

- Note that SQL Developer may allow execution of block with no ending / once the run is clicked. Details about use of SQL Developer is left for students to explore further.

PL/SQL Development Environment : SQLPLUS

- The SQL Buffer
  - Sqlplus stores recently typed SQL command or PL/SQL block in an area of memory called SQL buffer.
  - The SQL buffer remains unchanged until a new command is entered or you exit Sqlplus.
  - The SQL buffer can be edited by typing EDIT at SQL prompt.
  - While SQL and PL/SQL stmts are captured in the SQL buffer, Sqlplus commands (e.g., SET LINE…) are not captured in the buffer.

PL/SQL Development Environment : SQLPLUS

- When you create stored procedures, functions or packages, you begin with CREATE command.
- When you begin a PL/SQL block, you start by entering the word DECLARE or BEGIN
- Typing either BEGIN, DECLARE or CREATE puts the Sqlplus into PL/SQL mode.

- Running PL/SQL Blocks in Sqlplus
- How to End a PL/SQL block in Sqlplus
  - A PL/SQL block is ended with a period (.).

PL/SQL Development Environment : SQLPLUS

- How to Execute a PL/SQL Block in Sqlplus
  - A PL/SQL block is executed with a forward slash (/) or RUN
  - A PL/SQL program can be edited in sqlplus using EDIT
  - A PL/SQL program can be saved as a script file with a .sql extension. In that case, the file should be ended with a period to mark end of program, and followed with a forward slash (/) to execute the program when loaded.
  - To execute a script file in PL/SQL, use @filename.sql
- E.g., sql>@scriptfile.sql

Part B: Oracle Database Development (Oracle PL/SQL)

- **PL/SQL in Client/Server Architecture**
  - **Oracle applications can be built using client-server architecture where the Oracle database resides on the server and the program that requests data and changes on the database resides on a client machine.**
  - **The client program can be written in C, Java or PL/SQL**
  - **PL/SQL is not a stand-alone programming language like C or Java, but is part of the Oracle RDBMS.**
  - **PL/SQL can reside in two environments – client side and server side.**
  - **PL/SQL blocks are processed by PL/SQL engine, a special component of such Oracle products as Oracle server, Oracle Forms, Oracle Reports.**
  - **The SQL processor resides only on the Oracle server.**

PL/SQL Formatting Guide

- PL/SQL Formatting Guide
- CASE
- PL/SQL is case-insensitive [use upper case for Reserved keywords and lower case for others].
- WHITE SPACE
- Use proper indentation for readability.
- NAMING CONVENTIONS
- Use appropriate prefixes to distinguish identifiers standing for variables (eg, v_studentid), cursor (c_studentid), record (r_studentid), table (t_studentid), exception(e_studentid), etc.

Oracle PL/SQL

- PL/SQL processor sends SQL statements to the SQL processor to process when encountered.

- The PL/SQL Block Structure
  - The most basic unit in PL/SQL is a block
  - All PL/SQL programs are combined into blocks that are nested within each other.
  - PL/SQL blocks can be named or anonymous.
  - Named blocks are used for subroutines (which are procedures, functions and packages)
  - PL/SQL block has 3 sections: declaration section (optional), executable section (mandatory) and exception – handling section (optional).

Part B: PL/SQL IN A WRAP (slide 1 of 6)

- PL/SQL Program or block has a type and a structure as:
  - T: PL/SQL   block Type
  - S: PL/SQL block Structure
- T: PL/SQL   block Type
  - T1: Anonymous block (e.g.,)
  - T2: Named block
    - T2.1: Procedure (e.g.,)
    - T2.2: Function (e.g.,)
    - T2.3: Package (e.g.,)

Part B: PL/SQL IN A WRAP (slide 2 of 6)

- **S: PL/SQL block Structure**
  - **S1: Declaration section (optional)(e.g.)**
    - **S1.1: Data types and rules (e.g.,;)**
    - **(Varchar2, char, Number, binary_integer, Date, BOOLEAN, Long or CLOB, Rowid, %TYPE, Exception, %ROWTYPE, CURSOR, Type Record, Type Table, and Bfile or BLOB.**
    - **S1.2: Substitution variable for reading from the keyboard (e.g.,)**
    - **S1.3: Declaring Anchored Types ()**
    - **S1.4: Declaring Record Types:**
      - **S1.4.1. Cursors ()**
      - **S1.4.2. Using %ROWTYPE**
      - **S1.4.3. Using TYPE (like struct)**

Part B: PL/SQL IN A WRAP (slide 3 of 6)

  - S1.5: Declaring Exceptions ()
  - S1.6: Pl/SQL Table (arrays) ()
- S2. Executable Section
  - S2.1: SQL statements ()
  - S2.2: Printing instruction ()
    - DBMS_OUTPUT.PUTLINE(parameter);
  - S2.3: Assignment instructions ()
  - S2.4: Decision instructions (..)
    - S2.4.1: IF-THEN-ENDIF statement
    - S2.4.2: IF-THEN-ELSE-ENDIF statement
    - S2.4.3: IF-THEN-ELSIF----ELSE-ENDIF statement

Part B: PL/SQL IN A WRAP (slide 4 of 6)

- S2.5: Repetition instructions ()
  - S2.5.1: LOOP……END LOOP; statement
  - S2.5.2: FOR loop_counter IN [REVERSE] lower_limit .. Upper_limit LOOP ……. END LOOP; statement
  - S2.5.3: CURSOR FOR LOOP statement
  - S2.5.4: FOR UPDATE CURSOR statement
  - S2.5.5: WHILE condition LOOP …. END LOOP;
- S2.6: Declaraing and Calling a function, procedure or package ()
- S2.7: Declaring and calling a trigger ()

Part B: PL/SQL IN A WRAP (slide 5 of 6)

- **(Note1: expressions are important parts of all these instructions and substitution variables can be used in expressions).**
- **Note2: A function, procedure, or package must be declared, compiled successfully into p-code and stored in the database server as database object to be called by other program units.**

Part B: PL/SQL IN A WRAP (slide 6 of 6)

- **S3: Exception Handling Section ()**
  - **S3.1: Builtin exceptions**
    - **(VALUE_ERROR, NO_DATA_FOUND, TOO_MANY_ROW, ZERO_DIVIDE, LOGIN_DEFINED, PROGRAM_ERROR, DUP_VALUE_ON_INDEX)**
  - **S3.2: User Defined exceptions (e.g.,)**
    - **These must be declared in the declaration part, condition to raise them specified in the executable section and action to take when they occur specified in the exception handling section.**

Oracle PL/SQL: Structure of a block

Structure of an anonymous PL/SQL block is:

```
 DECLARE
   Declaration statements
 BEGIN
   Executable statements
EXCEPTION
   Exception-handling statements
END;
```

PL/SQL: Declaration Section

- Declaration section is for definitions of PL/SQL identifiers (variables, constants, cursors, etc)
- E.g.,

```
DECLARE
        v_first_name      VARCHAR2(35);
        v_last_name       VARCHAR2(35);
        v_counter         NUMBER:=0;
```

- A semicolon ends each declaration
- A variable declaration has the format

  identifier-name identifier-type (size);
- A constant CONSTANT declaration has the format
- constant-name CONSTANT -type := initial value;

PL/SQL: Executable Section

- Executable section starts with BEGIN statement and ends with END statement as in:

```
BEGIN
        SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
        WHERE student_id = 123;
        DBMS_OUTPUT.PUT_LINE
(‘Student name:’ || v_first_name || ‘ ‘ || v_last_name);
END;
```

PL/SQL: Executable Section

- Above selects first and last names of student with id 123 from db student table into PL/SQL variables v_first_name and v_last_name so that they can be printed using DBMS_OUTPUT.PUT_LINE statement.
  - An example Exception handling section for the above block is:

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
    ('There is no student with id 123');
```

PL/SQL: Reading Data with Substitution variables

- **Reading Data with Substitution variables**
  - **PL/SQL cannot accept input from a user directly.**
  - **However, sqlplus enables PL/SQL to receive input information with substitution variables.**
  - **Substitution variables are usually prefixed by the ampersand (&) or double ampersand (&&) character.**
  - **Substitution variables cannot be used to output values since no memory is allocated for them**

PL/SQL: Reading Data with Substitution variables

- **E.g., The following block prompts user for v_student_id (the substitution variable), which it stores as PL/SQL variable v_student_id. Then, it stores the first and last names of the student with this student id from student table in the database and displays the student names as output.**

PL/SQL: Reading Data with Substitution variables

```
DECLARE
      v_student_id NUMBER := &sv_studentid;
      v_first_name VARCHAR2(35);
      v_last_name  VARCHAR2(35);
BEGIN
      SELECT first_name, last_name
          INTO v_first_name, v_last_name
        FROM student
      WHERE student_id = v_student_id;
      DBMS_OUTPUT.PUTLINE
     ('Student Name: ' || v_first_name || ' ' || v_last_name);
EXCEPTION
      WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUTLINE ( 'No such student');
END;
```

PL/SQL: Reading Data with Substitution variables

- **When a single ampersand is used in a substitution variable, the user is prompted to enter a new value for each occurrence of the variable.**
- **E.g., on use of single substitution (&) variable**

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Today is ' || ' &sv_day');
    DBMS_OUTPUT.PUT_LINE ('Tomorrow is '|| '&sv_day');
END;
```

- **The above block produces the following output (last two lines)**
  - **Enter value for sv_day : Monday**
- **Old 2: DBMS_OUTPUT.PUT_LINE ('Today is' || '&sv_day');**
- **New 2: DBMS_OUTPUT.PUT_LINE ('Today is' || 'Monday');**

PL/SQL: Reading Data with Substitution variables

  - Enter value for sv_day: Tuesday
- Old 3: DBMS_OUTPUT.PUT_LINE (‘Tomorrow is’ || ‘&sv_day’);
- New 3: DBMS_OUTPUT.PUT_LINE (‘Tomorrow is’ || ‘Tuesday’);
- Today is Monday
- Tomorrow is Tuesday
- PL/SQL procedure successfully completed.
  - The program output contains statements showing how the substitution for the substitution variables are done. (e.g., statements beginning with old 2, new 2, old 3, new 3)

PL/SQL: Reading Data with Substitution variables

- To block the display of substitution statements, use the SET command option before running the script as in:

▪ SET VERIFY OFF;

- This gives the output that excludes the 4 statements beginning with old and new.
- When we use a substitution variable that is preceded by a double (&&), PL/SQL processor prompts the user to enter the value of this variable once first time used. Then, it substitutes this value for other uses of this variable (which should be single (&)) in the block.

PL/SQL: Reading Data with Substitution variables

- E.g., Use of Double (&&) substitution variable.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Today is' || '&&sv_day');
    DBMS_OUTPUT.PUT_LINE ('Tomorrow is' || '&&sv_day');
END;
```

- Here, user is prompted only once and if entered day is 'Monday', both output lines use Monday and result is like:

- Today is Monday
- Tomorrow is Monday
- PL/SQL procedure successfully completed.

PL/SQL: Reading Data with Substitution variables

- It is a good practice to enclose a substitution variable in single quotes if it is assigned to string (text) datatype as follows.
- E.g., Use of string substitution variable

  v_course_no VARCHAR2(5) := '&sv_course_no';
- Sqlplus allows changing the substitution variable character from (&) to a non-alphanumeric character specified using the following SET option

  SET DEFINE character

  SET DEFINE *
- To disable substitution variable feature, use:

  SET DEFINE OFF
- To enable substitution variable feature, use:

  SET DEFINE ON

PL/SQL: Displaying Output

- DISPLAYING OUTPUT with DBMS_OUTPUT.PUT_LINE
- The DBMS_OUTPUT.PUT_LINE is a call to procedure PUT_LINE in the DBMS_OUTPUT package of the Oracle user SYS
- This procedure DBMS_OUTPUT.PUT_LINE writes lines to buffer so that they can be displayed on the screen at the end of the program.
- The size of the buffer can be set to between 2000 and 1M bytes.
- Before output printed on the screen can be viewed, one of the following statements must be entered before the PL/SQL block.

```
SET SERVEROUTPUT ON;
        or
SET SERVEROUTPUT ON SIZE 5000;
```

PL/SQL: Displaying Output

- Both statements enable the DBMS_OUTPUT_PUTLINE statements. And while the first statement uses default buffer size, the second uses buffer size of 5000 byte.
- To disable info from being displayed on the screen, use:

  SET SERVEROUTPUT OFF;
- E.g., PL/SQL code for Exercise 1 on page 48 for computing the area of a circle given the radius as substitution variable is next.

PL/SQL: Displaying Output

- **Solution:**

```
DECLARE
   v_radius NUMBER := &sv_radius;
   v_area    NUMBER := v_radius * v_radius * 3014;
BEGIN
  DBMS_OUTPUT.PUT_LINE ( ‘Area of Circle with
  radius’ || v_radius || ‘is’      || v_area);
END;
```

PL/SQL Programming Fundamentals

- PL/SQL Programming Fundamentals
- Character Types
  - PL/SQL engine accepts four types of characters (letters, digits, symbols (*, +, -, =, …) and white space.
  - Combinations of characters form one of the valid 5 lexical units (identifiers, reserved words, delimiters, literals, comments).
  - Identifiers begin with a letter and can be up to 30 characters long (avoid reserved words).
  - Reserved words like BEGIN, END etc are for use by PL/SQL
  - Delimiters are arithmetic, comparison and logical operators and quotation marks.

PL/SQL Programming Fundamentals

- Literals are values that are not identifiers, e.g., 150, 'Holiday', FALSE.
- Comments: lines beginning with (--) are single line comments while those lines between (/*) and (*/) are multiple line comments.

Anchored Datatypes.

- An anchored datatype is based on the datatype of a database object (like database attribute, e.g., student.firstname).
- Giving a PL/SQL variable, an anchored datatype that is similar to the datatype of database attribute, Student.first_name can be done with the following instruction:

v_name student.first_name%TYPE;

- General syntax for declaring variable of anchored type is:

<variable_name> <type attribute> % TYPE;

PL/SQL Programming Fundamentals

- E.g.,

```
DECLARE
    v_name student.first_name % TYPE;
    v_grade grade.grade_type_code % TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE (NVL(v_name, ‘No
    Name’) || ‘ has grade of  ’ ||  NVL(v_grade, ‘ no grade’);
END;
```

PL/SQL Programming Fundamentals

- **DECLARING AND INITIALIZING VARIABLES**
  - **Each variable declared to be used by the program in the DECLARATION section should be terminated with a semicolon.**
  - **A numeric constant variable must be assigned a value with (:=) at declaration time and this value cannot be changed later in the program**
  - **A constant variable during declaration includes the keyword CONSTANT as in:**

```
v_cookies_calorie CONSTANT NUMBER := 300;
```

PL/SQL Programming Fundamentals

- Example declarations are:

```
DECLARE
    v_lname VARCHAR2(30)
    v_regdate DATE;
    v_pctincv CONSTANT NUMBER(4, 2) := 1.15;
    v_counter NUMBER := 0;
    v_new_cost course.crsecost % TYPE;
    v_yorn BOOLEAN := TRUE;
BEGIN
     NULL;
END;
```

EXPRESSIONS, OPERANDS AND OPERATORS

- An expression is a sequence of variables and literals, separated by operators, for performing calculatioins and comparing data.
- An expression is a combination of operands and operators.
- An operand is a variable, a constant or a function call.
- An operator is arithmetic (**, /, *, +, -), comparison (<, >, <>, =, >=, <=, !=, like, in, between, is null), logical (AND, OR, NOT), string (||, like)
- Parentheses can be used to enforce the order of execution of an expression.
- General operator precedence is
  - **, NOT,
  - +, -, arith identity and negation, *, /, +, -, ||, =, <>, <=, <, >, like, between, IN, IS NULL.
  - AND
  - OR

EXPRESSIONS, OPERANDS AND OPERATORS

- E.g., expressions are:

((v_counter + 5) * 2) / 2

(v_new_cost * v_counter) / 5

- Expressions form the right sides of assignment instructions like:

v_counter := ((v_counter +5) * 2) / 2;

v_new_cost := (v_new_cost * v_counter) / 4;

Use of Labels, Scope of Block & Variables

- Use of Labels
  - Labels can be used for readability and label for a block must appear before the first line of executable code (BEGIN or DECLARE) as follows.

```
<<find_stu_num>>
BEGIN
    DBMS_OUTPUT.PUT_LINE('procedure find_stu_num has been executed.');
END find_stu_num;
```

- Scope of a Block & Variables
  - The scope or existence of variables defined in the declaration section of a block is the block.
  - A nested block is a block totally inside another block.

Scope of Block & Variables; Common Data Types

- Visibility of a variable is the part of the program where this variable can be used or accessed.
- Scope of exception is also the block it is defined.

- Most Common Datatypes
- 1. VARCHAR2 (maximum_length): takes character variable specifying maximum length of up to 3276 bytes. Maximum width of a VARCHAR2 database column is 2000 bytes.
- 2. CHAR (maximum_length): stores fixed size character with specified MAX_length, that is possibly padded with blanks. Maximum length that can be specified is 32767 bytes although maximum length of a database column that can be stored with this type is 255 bytes. Default length is set to 1 if max_length is not specified.

Common Data Types

- 3. NUMBER [(precision, scale)]: stores fixed or floating-point number of any size where precision represents number of digits and scale determines number of digits following decimal point.
  - When scale is omitted, it represents integer number
  - Maximum precision is 38 decimal digits
  - A negative scale causes rounding to the left of the decimal point.
  - E.g., with the declarations

    v_num NUMBER (6, 2) := 3.456;

    v_num NUMBER (6, 3) := 3456;

    v_num has 3.46 and v_numl has 3000.
  - When scale is not specified, it defaults to 0 (rounding to nearest whole number).
- 4. BINARY INTEGER: stores signed integer variables in binary format for less space and more efficiency.

Common Data Types

- 5. DATE: stores fixed_length date values from January 1, 4712 BC to December 31, 4712 AD.
  - When stored in database column, date values include the time of day in seconds since midnight. The date portion defaults to midnight. Dates are displayed according to default format.
- 6. BOOLEAN: stores the values TRUE and FALSE and the non-value NULL. The values TRUE and FALSE cannot be inserted into a database column.
- 7. LONG: stores variable-length character strings of up to 32, 760 bytes, and can be inserted into a LONG database column, (which has a maximum width of 2, 147,483,647 bytes.
  - We cannot select a value longer than 32, 760 bytes from a LONG column into a LONG variable.
  - LONG columns can store text arrays of characters, or short documents, can be referenced in UPDATE, INSERT and (most) SELECT statements but not in expressions, SQL function calls, or certain SQL clauses such as WHERE, GROUP BY and CONNECT BY.
- 8. ROWID: stores rowids in a readable format. Internally, every Oracle database table has a ROWID pseudo column, which stores binary values called rowids.

Managing PL/SQL Code with SQL

- Managing PL/SQL Code with SQL
  - The changes to the database due to an application session are saved into the database after a COMMIT is executed.
  - Work within a transaction up to commit can be ROLLED BACK (that is undone).
  - A transaction is a series of SQL statements grouped together into a logical unit by the programmer.
  - A SAVEPOINT can be used to break down large SQL statements into individual units easier to manipulate.

Variable Initialization

- Variable Initialization with SELECT INTO
  - In PL/SQL, variables can be assigned values in one of 2 ways:
    - During declaration with ':='
    - Assigning a value with SELECT INTO statement.
- SELECT INTO Statement: The Syntax of assignment with SELECT INTO is:

```
SELECT item_name
INTO variable_name
FROM table_name;
```

Variable Initialization

- E. g.,

```
SET SERVEROUTPUT ON;
DECLARE
    v_average_cost VARCHAR2(10);
BEGIN
    SELECT  To_char (Avg(cost), '$9, 999. 99')
    INTO  v_average_cost
     FROM  course;
    DBMS_OUTPUT.PUT_LINE(' The average cost of a '||' course in
    the CTA program is '|| v_average_cost);
END;
```

Variable Initialization

- Variable v_average_cost is given the datatype VARCHAR2 because of the function used on the data.
- The TO_CHAR function formats the cost and the number datatype is converted to a character datatype.
- Another example of use of DML statement in PL/SQL block is:

```
DECLARE
    v_city zipcode.city % TYPE;
BEGIN
    SELECT 'COLUMBUS'
    INTO v_city
    FROM dual;
    UPDATE zipcode
        SET city = v_city
   WHERE zip = 43224;
END;
```

Inserting Data in PL/SQL

- DDL is not valid in a simple PL/SQL block.
- Data can be inserted as shown in the following example.

```
DECLARE
    v_zip zipcode.zip % TYPE;
    v_user zipcode.created_by % TYPE;
    v_date zipcode.created_date % TYPE;
BEGIN
   SELECT 43438, USER, SYSDATE
   INTO v_zip, v_user, v_date
    FROM dual;
   INSERT INTO zipcode
   (ZIP, CREATED_BY, CREATED_DATE, MODIFIED_BY,
   MODIFIED_DATE)
   VALUES (v_zip, v_user, v_date, v_user, v_date);
END;
```

Using an Oracle Sequence

- USING AN ORACLE SEQUENCE
  - An Oracle sequence is a database object used to generate unique numbers like primary keys.
  - Already created sequence values in SQL statements can be accessed with pseudo columns.
- CURRVAL (for returning the sequence current value)
- NEXTVAL (for incrementing the sequence and returning new value)
  - E.g., to create a sequence called ESEQ in sqlplus, we use:
- CREATE SEQUENCE eseq INCREMENT BY 10;
  - This sequence can be used to populate the column number attribute of a table called Teacher as follows:

Using an Oracle Sequence

```
CREATE SEQUENCE ESEQ
   INCREMENT BY 10;
CREATE TABLE TEACHER (col number);
BEGIN
      INSERT INTO TEACHER
       VALUES (ESEQ.NEXTVAL);
END;
```

Making Use of Savepoint

- Making Use of SavePoint
  - A transaction is a logical unit of work consisting of a set of SQL statements.
  - A transaction would either succeed (once a COMMIT is executed) or fail (if not successfully committed) as a unit.
  - The PL/SQL block for one transaction ends with COMMIT or ROLLBACK.
  - COMMIT makes events within a transaction permanent and releases all locks required by the transaction.
  - ROLLBACK erases (undoes) events within a transaction and releases all locks acquired by transaction.

Making Use of Savepoint

- SAVEPOINT can be used to control transaction such that SQL statements are split into transaction units that can be committed and rolled back as necessary.
- A COMMIT statement has the syntax:
- COMMIT [WORK];
- The word WORK is optionally used for readability.
- A ROLLBACK statement has following syntax:
- ROLLBACK [WORK];
- A SAVEPOINT command has the following syntax:
- SAVEPOINT name;
- The word name is the SAVEPOINT's name.

Making Use of Savepoint

- A program can be made to rollback to a SAVEPOINT using the more general form of ROLLBACK instruction below:

- ROLLBACK [WORK] to SAVEPOINT name;

- E.g., Page 81-82

```
BEGIN
  INSERT INTO student
     (student_id, Last_name, zip, registration_date, created_by,
     created_date,   modified_by, modified_date)
  VALUES (student_id_seq.nextval, 'Tashi', 10015, '01-JAN-99',
     'STUDENTA', '01-JAN-99', 'STUDENTA', '01-JAN-99'    );
 SAVEPOINT A;
```

Making Use of Savepoint

```
INSERT INTO student
  (student_id, Last_name, zip, registration_date, created_by, created_date
,  modified_by, modified_date)
  VALUES (student_id_seq.nextval, 'Sonam', 10015, '01-JAN-99',
'STUDENTB',
 '01-JAN-99', 'STUDENTB', '01-JAN-99');
  SAVEPOINT B;
  INSERT INTO student
  (student_id, last_name, zip, registration_date, created_by, created_date
, modified_by, modified_date)
  VALUES (student_id_seq.nextval, 'Norbu', 10015, '01-JAN-99',
  'STUDENTA', '01-JAN-99', 'STUDENTB', '01-JAN-99');
  SAVEPOINT C;
   ROLLBACK TO B;
  END;
```

Making Use of Savepoint

- An example PL/SQL block that can contain multiple transactions

```
DECLARE
    v_counter NUMBER;
BEGIN
    v_counter := 0;
    FOR  i IN  1 .. 100
    LOOP
        v_counter := v_counter + 1;
        IF v_counter = 10
        THEN
            COMMIT;
            v_counter := 0;
        END IF;
    END LOOP;
END;
```

- **Here, when v_counter hits 10, it commits keeping 10 transactions in one PL/SQL block.**

Types of Instructions

- 1. Assignment Instructions
  - 1.1 Using assignment operator :=
    E.g., v_counter := ((v_counter + 5)) * 2)/2;
  - 1.2 Using SQL statements like:
    SELECT first_name, last_name
    INTO v_firstname, v_lastname
    FROM STUDENT
    WHERE stuid = v_stud_id;
- 2. Print and Read statements
  - 2.1 Print instructions with
    DBMS_OUTPUT.PUT_LINE as in:

Types of Instructions

- DBMS_OUTPUT.PUT_LINE('Area of Circle is' || v_area);
- 2.2. Read from the keyboard with substitution variables as in e.g.,
  NUMBER := &sv_radius;

- 3. Conditional Instructions (see slides 108 to 116 for IF statement examples)
  - 3.1 IF-THEN statement
  - 3.2 IF-THEN-ELSE statement
  - 3.3. IF-ELSIF. ….ELSE statement
  - 3.4 CASE statements: CASE form is given next.

Types of Instructions

- CASE condition
```
    WHEN expression 1 THEN statement 1;
    WHEN expression 2 THEN statement 2;
    ………
    WHEN expression N THEN statement N;
    ELSE statement N+1;
END CASE;
```

Types of Instructions

- 4. Repetition Instructions
  - 4.1 Simple Loop (LOOP …. END LOOP)
  - 4.2 Numeric FOR LOOP (FOR loop_counter IN [REVERSE] lower_limit .. Upper_limit LOOP ….. END LOOP;]
  - 4.3 Variations of FOR loop used for CURSOR (CURSOR FOR LOOP …)
  - 4.4 WHILE condition LOOP (WHILE condition LOOP …. END LOOP)

Conditional Control

- Conditional Control
- IF Statements
- An IF-THEN statement has the following structure:

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

Conditional Control

- If the CONDITION expression evaluates to TRUE, statements 1 to N are executed.
- E.g., write a PL/SQL block that compares two integer values in v_num1 and v_num2 and stores the smaller value always in v_num1.

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

Conditional Control

```
    THEN
      v_temp := v_num1;
      v_num1 := v_num2;
      v_num2 := v_temp;
    END IF;
      DBMS_OUTPUT.PUT_LINE(‘v_num1=’ || v_num1);
      DBMS_OUTPUT.PUT_LINE(‘v_num2=’ || v_num2);
END;
```

- The above produces the following output

  v_num1 = 3

  v_num2 = 5
- PL/SQL procedure successfully completed.

Conditional Control

- IF-THEN-ELSE STATEMENT
- The structure of the IF-THEN-ELSE statement is:

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

- STATEMENT 3;
- When CONDITOIN evaluates to TRUE, STATEMENT 1 is executed, but if it is FALSE, STATEMENT 2 is executed. The next statements in the program executed

Conditional Control

after the IF-THEN-ELSE statement is STATEMENT 3. E.g, Use of IF-THEN-ELSE statement is shown next.

```
DECLARE
        v_num NUMBER := &sv_user_num;
BEGIN
        -- test if provided number is even
IF MOD (v_num, 2) = 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num || ‘is even’);
    ELSE
   DBMS_OUTPUT.PUT_LINE (v_num || ‘is odd’);
   END IF;
DBMS_OUTPUT.PUT_LINE (‘Done’);
END;
```

NULL Condition

- A NULL condition may arise if one of the compared variables has no value, for example:

```
DECLARE
    v_num1 NUMBER := 0
    v_num2 NUMBER;
BEGIN
   IF v_num1 = v_num2
   THEN
        DBMS_OUTPUT.PUT_LINE('They are equal');
   ELSE
        DBMS_OUTPUT.PUT_LINE('They are not equal');
   END IF;
END;
```

NULL Condition

- Note that v_num2 has no value leading to a NULL condition that evaluates to NULL and treated as false in this case.
- Use of Some Functions (Page 95), eg.
- TO_DATE, TO_CHAR, RTRIM

  v_date DATE := TO_DATE('&sv_user_date', 'DD-MM-YY');

  v_day := RTRIM(TO_CHAR(v_date, 'DAY'));
- In the above instructions, the function TO_CHAR returns the day of the week with v_date padded with blanks since this function always returns 9 bytes.
- Next, the function RTRIM is used to remove trailing spaces.

ELSIF STATEMENT

ELSIF statements

An Elsif statement has the following structure

```
IF CONDITION 1
    THEN
  STATEMENT 1;
ELSIF CONDITION 2
    THEN
  STATEMENT2;
ELSIF CONDITION 3
    THEN
  STATEMENT 3;
…
ELSE
    STATEMENT N;
END IF;
```

Condition Control

- Only one of statements 1 to N is executed depending on which of conditions 1 to N evaluates to TRUE. E.g.,

```
DECLARE
        v_num NUMBER := &sv_num;
BEGIN
        IF v_num < 0
        THEN
            DBMS_OUTPUT.PUT_LINE ( v_num || ‘is a negative number’);
        ELSIF v_num = 0
      THEN
            DBMS_OUTPUT.PUT_LINE ( v_num || ‘is equal to zero’);
      ELSE
            DBMS_OUTPUT.PUT_LINE ( v_num || ‘is a positive number’);
        END IF;
END;
```

Exception Handling

- Exception Handling and Builtin Exception
  - Exception handling section in a PL/SQL block specifies what action to take when an exception error occurs.
  - Two types of exceptions exist – builtin and user-defined exceptions
  - Errors that occur in a program are either compilation or runtime errors. Exceptions are defined mostly for runtime errors.
  - Compilation errors are due to language syntax violation and are also called syntax errors.
- E.g., v_num1 = v_num1 / v_num2;

Exception Handling

- Will generate syntax error because assignment operator is (:=) and not (=). Statement should then be changed to following and re-compiled:
- v_num1 := v_num1 / v_num2;
  - Assume v_num1 has an initial value of 5 while v_num2 has a value of 0. Running this statement in a PL/SQL block leads to a runtime error because an illegal operation of dividing by zero has occurred.
  - Compilation and runtime errors may cause the program to not successfully complete.
  - Exception handling section is used to produce informative message when a runtime exception occurs. It also makes the program end successfully.

Exception Handling

- Exceptions:
  - VALUE_ERROR: This is raised when there is a conversion or size mismatch error. Eg, v_num := SQRT(v_num1); if v_num1 has a negative value, the SQRT function cannot accept it, raising a VALUE_ERROR.
- Usage:

```
EXCEPTION
      WHEN VALUE_ERROR THEN
                DBMS_OUTPUT.PUT_LINE('Value Error Occurs');
```

- NO_DATA_FOUND: raised when a select into statement, which makes no calls to group functions such as SUM or COUNT, does not return any rows. [Note that if the select makes a call to a group function like count, if nothing is found, it returns 0, and thus there is no need to raise a NO_DATA_FOUND exception in that case].
- TOO_MANY_ROWS: raised when a SELECT INTO statement returns more than one row [It normally should return only one row].

Exception Handling

- **ZERO_DIVIDE: raised when a division by zero is performed.**
- **LOGIN_DEFINED: raised when a user is trying to log on to Oracle with invalid username and password.**
- **PROGRAM_ERROR: raised when the PL/SQL program has an internal problem.**
- **DUP_VALUE_ON_INDEX: raised when a program tries to store a duplicate value in the columns that have unique index defined on them. E.g., inserting values for course #, section # for course Comp 4150, section 1 that already exists and has a unique index defined on it.**

Cursors

- **Example use of Cursor**

```
DECLARE
v_sid student.student_id%TYPE
CURSOR c_student IS
SELECT student_id
FROM student
WHERE student_id < 110;
BEGIN
     OPEN c_student;
     LOOP
        FETCH c_student INTO v_sid;
        EXIT WHEN c_student % NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(‘STUDENT ID:’ || v_sid);
```

Cursors

```
END LOOP;
CLOSE c_student;
EXCEPTION
  WHEN OTHERS
  THEN
        IF c_student % ISOPEN
        THEN
              CLOSE c_student;
        END IF;
END;
```

Cursors

- **Using Cursor For LOOPS and Nesting Cursors**
- **Cursor FOR LOOP statement opens, fetches, and closes the cursor implicitly.**
- **The cursor FOR LOOP specifies a sequence of statements to be repeated once for each row returned by the cursor.**
- **Use the cursor FOR LOOP if you need to FETCH and PROCESS each and every record from a cursor.**

Cursors

- **For example, assume the existence of a table called table_log with one column.**

```
DECLARE
    Cursor c_student IS
    SELECT student_id, last_name, first_name
    FROM student
    WHERE student_id < 110;
BEGIN
    FOR r_student IN c_student
    LOOP
      INSERT INTO table_log
      VALUES (r_student.last_name);
    END LOOP;
END;
```

Cursors

- Cursors can be nested inside each other
- Example nested cursor with a single child cursor.

```
DECLARE
    v_zip zipcode.zip % TYPE;
    CURSOR c_zip IS
        SELECT zip, city, state
        FROM zipcode
        WHERE state = ‘CT’;
    CURSOR c_student IS
        SELECT first_name, last_name
        FROM student
        WHERE zip = v_zip;
```

Cursors

```
BEGIN
        FOR r_zip IN c_zip
        LOOP
                v_zip := r_zip.zip;
                DBMS_OUTPUT.PUT_LINE(CHR(10));
                DBMS_OUTPUT.PUT_LINE(‘Students living in’
    ||r_zip.city);
                FOR r_student IN c_student
                LOOP
                DBMS_OUTPUT.PUT_LINE(r_student.first_name || ‘ ‘ ||
                        r_student.last_name);
                END LOOP;
        END LOOP;
    END;
```

Cursors

- USING PARAMETERS WITH CURSORS AND FOR UPDATE CURSORS
- A cursor can be declared with parameters to enable it generate a more specific result set and make itself more reusable.
- E.g., create a cursor that works for only a set of values.

```
CURSOR c_zip (p_state IN zipcode.state % TYPE)
IS
SELECT zip, city, state
FROM ZIPCODE
WHERE state = p_state;
```

Cursors

- A cursor declared to take a parameter must be called with a value for that parameter.
- The c_zip cursor is called as follows:

  OPEN c_zip (parameter_value);

  OPEN c_zip (‘NY’);
- Using a FOR UPDATE CURSOR
- The cursor FOR UPDATE clause is only used with a cursor when you want to update tables in the database.
- This entails simply adding FOR UPDATE to the end of the cursor definition.
- Using the FOR UPDATE has the effect of locking the rows that have been identified in the active set.

Cursors

- If we want to lock only one of multiple items being selected, add: FOR UPDATE OF <item_name>
- E.g.,

```
DECLARE
    CURSOR c_course IS
      SELECT course_no, cost
      FROM course FOR UPDATE;
BEGIN
    FOR r_course IN c_course
    LOOP
          IF r_course.cost < 2500
          THEN
```

Cursors

```
UPDATE course
    SET crsecost = r_course.cost + 10
    WHERE course_no = r_course.course_no;
    END IF;
  END LOOP;
END;
```

- WHERE CURRENT OF CLAUSE
- WHERE CURRENT OF <cursor_name> can be used to update the most recently fetched row as in:

Cursors

```
DECLARE
    v_zip zipcode.zip % TYPE;
    CURSOR c_student IS
        SELECT student_id, first_name, last_name, zip, phone
        FROM student
        FOR UPDATE;
BEGIN
        FOR r_stud_zip IN c_student
        LOOP
            DBMS_OUTPUT.PUT_LINE(r_stud_zip.student_id);
            UPDATE student
            SET  phone = '718' || SUBSTR(phone, 4)
            WHERE CURRENT OF c_student;
        END LOOP;
END;
```

User-Defined Exceptions

- User-Defined Exceptions
- Exceptions can be defined by programmer and must first be declared with the following syntax:

```
DECLARE
    Exception_name EXCEPTION;
```

- The executable statements of a user declared exception are specified in the exception-handling section of the block.
- E.g., use of user defined exception

User-Defined Exceptions

```
DECLARE
    e_invalid_id EXCEPTION;
BEGIN
    WHEN e_invalid_id
    THEN
      DBMS_OUTPUT.PUT_LINE('A negative id is not allowed');
END;
```

- User defined exceptions have to be raised explicitly by defining what conditions should cause them to be triggered.
- How is given below:

User-Defined Exceptions

```
DECLARE
    Exception_name EXCEPTION;
BEGIN
……
    IF CONDITION
    THEN
       RAISE exception_name;
    ELSE
       ……
    END IF;
EXCEPTION
    WHEN exception_name
    THEN
       ERROR-PROCESSING STATEMENTS;
END;
```

User-Defined Exceptions

- Exception Propagation
- The rules governing how exceptions are raised in declaration and exception-handling sections are called Exception Propagation.
- When a runtime error occurs in the declaration or exception handling section, the exception handling section of this block is not able to catch it.
- In a program with nested PL/SQL blocks, when a runtime error occurs in the declaration section of the inner block, the exception immediately propagates to the enclosing outer block.

User-Defined Exceptions

- **Exception: Advanced Concepts**
- **Raise_Application_Error**
- **Raise_Application_Error is used to assign an exception number and message to a user_defined exception.**
- **The syntax of the use of this procedure is:**
  RAISE_APPLICATOIN_ERROR(error_number, error_message);
- **or**
  RAISE_APPLICATION_ERROR(error_number, error_message, keep_errors);
- **E.g.,**

User-Defined Exceptions

```
SET SERVEROUTPUT ON;
DECLARE
v_student_id  NUMBER := &sv_student_id;
v_total_courses NUMBER;
e_invalid_id EXCEPTION;

BEGIN
      IF v_student_id < 0
      THEN
            RAISE e_invalid_id;
      ELSE
            SELECT count(*)
            INTO v_total_courses
            FROM enrollment
```

User-Defined Exceptions

```
WHERE student_id = v_student_id;
            DBMS_OUTPUT.PUT_LINE('The student is registered
   for ' || v_total_courses || ' Courses');
      END IF;
END;
EXCEPTION
      WHEN e_invalid_id
      THEN
        DBMS_OUTPUT.PUT_LINE('The entered id is
   invalid');
END;
```

User-Defined Exceptions

- EXCEPTION_INIT_PRAGMA
- The EXCEPTION_INIT PRAGMA
- is used to associate an Oracle error number with a name of a user-defined error so that a handler may be written for it.
- The EXCEPTION_INIT pragma appears in the declaration section as:

```
DECLARE
    exception_name EXCEPTION;
        PRAGMA EXCEPTION_INIT(exception_name,
    error_code);
```

- The user_defined exception has to be declared before the EXCEPTION_INIT pragma that uses it.

User-Defined Exceptions

- SQLCODE and SQLERRM
- Oracle exception handler OTHERS can trap all Oracle errors.
- However, it is hard to know which error occurred if OTHER is used to trap it.
- Two built-in functions SQLCODE and SQLERRM can be used with the OTHERS exception handler to return the error number and message respectively.
- SQLERRM returns a message that is less than or equal to 512 bytes, while SQLCODE generally returns a negative error number.
- Example

User-Defined Exceptions

```
DECLARE
  v_zip VARCHAR2(5) := '&sv_zip';
  v_city VARCHAR2(15);
  v_state CHAR(2);
  v_err_code NUMBER;
  v_err_msg VARCHAR2(200);
BEGIN
        SELECT city, state
        INTO v_city, v_state
        FROM zipcode
        WHERE zip = v_zip;
        DBMS_OUTPUT.PUT_LINE(v_city || ' ' || v_state);
EXCEPTION
    WHEN OTHERS
    THEN
        v_err_code  := SQLCODE;
        v_err_msg := SUBSTR(SQLERRM, 1, 200);
        DBMS_OUTPUT.PUT_LINE('Error code: '|| v_err_code);
        DBMS_OUTPUT.PUT_LINE('Error message : '|| v_err_msg);
END;
```

Procedures

- SQLCODE, however returns a 0 if it is referenced outside the exception section, it returns +1 for user_defined exceptions and 100 for NO_DATA_FOUND exception.

- PROCEDURES
- Procedures allow structuring a program into modules (distinct subsolutions)
- Each module performs a specific task that contributes toward the final program goal.
- Modular code stored on database server is called a database object or subprogram that is available to other program units for repeated use.
- To save code into the database, it needs to be compiled into p-code and stored in database server.

Procedures

- A PL/SQL module is a complete logical unit of work and four types exist as:
- anonymous blocks
- procedures
- functions, and
- packages
- modular codes are more usable and manageable.

Procedures

- 1. ANONYMOUS BLOCKS
  - These have no names and no parameters.
  - Consists of Declaration, Execution and optional Exception parts.
  - They are not stored in the database as they cannot be called by other blocks. All examples before now are anonymous blocks.
- 2. PROCEDURES
  - A procedure may have 0 or more parameters and must have a name. The syntax of a procedure is:
- CREATE OR REPLACE PROCEDURE

```
CREATE OR REPLACE PROCEDURE
        name [(parameter1, parameter2, …)]
AS IS [local declarations]
BEGIN
        Executable statements
[EXCEPTION exception handlers]

END [name];
```

Procedures

- **A procedure consists of (1) the header [everything before the AS or IS keyword used interchangeably], (2) the body [everything after the AS or IS keyword].**
- **The word REPLACE is optional but if not used, changing procedure code will entail dropping and re-creating.**
- **E.g.,**

Procedures

```
CREATE OR REPLACE PROCEDURE Discount
AS
    CURSOR c_group_discount
IS
    SELECT distinct s.course_no, c.description
    FROM section s, enrollment e, course c
    WHERE s.section_id = e.section_id
    AND c.course_no = s.course_no
    GROUP BY s.course_no, c.description, e.section_id,
    s.section_id
    HAVING COUNT(*) >= 8;
```

Procedures

```
BEGIN
    FOR r_group_discount IN c_group_discount
    LOOP
            UPDATE course
            SET cost = cost * .95
            WHERE course_no = r_group_discount.course_no;
            DBMS_OUTPUT.PUT_LINE ('A 5% discount has been
    given to' || r_group_discount.course_no || ' ' ||
r_group_discount.description);
    END LOOP;
END;
```

Procedures

- To have the procedure update the database, a COMMIT needs to be issued after running the procedure (after END). It can also be placed after the END LOOP statement.
- A procedure can become invalid when the table it is based on is deleted or changed.
- To re_compile an invalid procedure, use:
- ALTER procedure procedure_name compile;

Procedures

- PROCEDURES AND DATA DICTIONARY
- Data dictionary provides information on stored procedures in either
  - USER_OBJECTS view (information about objects), or
  - USER_SOURCE view (source code text)
- Data dictionary also has an ALL_ and DBA_ version of these views.
- PASSING PARAMETERS IN AND OUT OF PROCEDURES
- Parameters are used to pass values to and from calling procedures to the server.
- Parameters are available in 3 modes as IN, OUT, and INOUT.
- Parameter mode specifies whether it is:

Procedures

- **IN: an input parameter that simply passes a value to the procedure for read only and this parameter cannot be changed by the procedure.**
- **OUT: an output parameter that passes result back from the procedure**
- **INOUT: both input and output parameter for passing value in and sending result back.**

Example Procedure with Parameters

```
CREATE OR REPLACE
    PROCEDURE FIND_NAME( ID IN NUMBER,  LNAME OUT
VARCHAR2,
    FNAME OUT VARCHAR2)       AS
        BEGIN
            SELECT last_name, first_name
            INTO LNAME, FNAME
             FROM student
              WHERE student_id = ID;
```

Procedures

```
EXCEPTION
        WHEN OTHERS
        THEN
        DBMS_OUTPUT.PUT_LINE('
  Student id not found ');
END FIND_NAME;
```

- In the example, the parameters ID and LNAME, FNAME in the procedure header are formal parameters

- Formal parameters are place holders for actual data values passed in or out with actual parameters during procedure call.

Procedures

- Formal parameters do not require datatype constraints like size, e.g.,
- VARCHAR2(60) is stated as VARCHAR2.
- When matching actual and formal parameters, use positional notation or named notation.
- Named notation associates formal parameter to its actual value during procedure call explicitly using the format: (formal parameter => actual parameter).
- Calling a Stored Procedure
- The procedure find_name defined above can be called in another anonymous block

Procedures

- as:

```
DECLARE
        ID student.student_id%TYPE;
        v_local_fname student.first_name%TYPE;
        v_local_lname student.last_name%TYPE;
BEGIN
        ID := 250;
        find_name(ID, v_local_lname, v_local_fname);
DBMS_OUTPUT.PUT_LINE('Student ' || ID || ' is
    '||v_local_fname || '  ' || v_local_lname);
END;
```

Functions

- FUNCTIONS
- Function is a PL/SQL procedure that returns a single value.
- Function definition structure is:

```
CREATE [OR REPLACE] FUNCTION
      function_name (parameter list)
RETURN datatype
IS
BEGIN
      <body>
      RETURN (return_value);
END;
```

- In a function, there should be a RETURN statement for each exception
- Function parameters can be of IN, OUT or INOUT types.

Functions

- E.g.,

```
CREATE OR REPLACE FUNCTION
Show_description(i_course_no NUMBER)
    RETURN VARCHAR2
AS
    v_description VARCHAR2(50);
BEGIN
    SELECT description
    INTO v_description
    FROM course
    WHERE course_no = i_course_no;
    RETURN v_description;
```

Functions

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
     RETURN ('The cursor is not in the database');

WHEN OTHERS
THEN
   RETURN ('Error in running show_description');
END;
```

- The function declared above can be invoked in the SELECT statement below:

Packages

```
SELECT course_no, show_description(course_no)
FROM course;
```

PACKAGES

- A collection of PL/SQL objects grouped together as a logical unit under one package name is called a package.
- Packages include procedures, functions, cursors, declarations, types and variables.
- First call to a package causes loading the package in memory, while subsequent calls save compilation and loading time.
- Packages encourage top down design and improve on information hiding and security of code.
- A package consists of Specification and Body, which may be compiled separately.

Packages

- Package Specification contains declaration information about objects in the package (procedures, functions and not their codes, global/public variables). All objects in a package specification are public objects.
- Private Procedures/Functions are not in the package specification but coded in its body.

```
CREATE OR REPLACE PACKAGE
   manage_students
AS
```

Packages

```
PROCEDURE FIND_NAME
( ID IN NUMBER,  LNAME OUT  VARCHAR2,
   FNAME OUT VARCHAR2);
FUNCTION id_is_good(i_student_id NUMBER)
RETURN BOOLEAN;
END manage_students;
```

- An example package specification consisting of a procedure and a function is given above.

Package Body

- Package Body
- The package body contains actual executable code of the objects described in the package specification
- Package body may contain additional code for private objects not declared in the specification of the package.
- The headers of the cursor and modules and their definitions in the package specification should match exactly.
- Elements declared in the specification can be referenced in the body and should not be re-declared.

Package Body

- Package elements can be referenced outside the package using the notation:
- package_name.element
- Elements referenced inside the body of the package do not need to be qualified.
- The package body of the above specification is:

```
CREATE OR REPLACE PACKAGE BODY manage_students
AS
PROCEDURE FIND_NAME
      (ID IN NUMBER,
      LNAME OUT student.last_name % TYPE,
      FNAME OUT student.first_name % TYPE)
IS
```

Package Body

```
BEGIN
    SELECT first_name, last_name
    INTO o_fname, o_lname
    FROM student
    WHERE student_id = ID;
    EXCEPTION
        WHEN OTHER
        THEN
            DBMS_OUTPUT.PUT_LINE('Error in finding
    student id:'|| ID);
END find_sname;
```

Package Body

```
    FUNCTION id_is_good
                ( i_student_id NUMBER)
    RETURN BOOLEAN
    IS
       v_id_cnt number;
    BEGIN
       SELECT COUNT(*)
       INTO v_id_cnt
       FROM student
       WHERE student_id = i_student_id;
       RETURN v_id_cnt=1;
    EXCEPTION
    WHEN OTHERS
    THEN
       RETURN FALSE;
    END id_is_good;
END manage_students;
```

Calling Stored Packages

- CALLING STORED PACKAGES
- The following anonymous block shows how elements of manage_student package are called by other blocks.

```
DECLARE
    v_first_name student.first_name % TYPE;
    v_last_name student.last_name % TYPE;
BEGIN
    IF manage_students.id_is_good (& v_id)
    THEN
      manage_students.find_sname
```

Calling Stored Packages

```
(&&v_id, v_first_name, v_last_name);
        DBMS_OUTPUT.PUT_LINE('Student No' || && v_id || 'is' ||
        v_last_name || ';' || v_first_name);
        ELSE
                DBMS_OUTPUT.PUT_LINE('Student ID' || &&
                v_id || 'is not in the database.');
        END IF;
END;
```

- Find out why actual parameter v_id is passed with & and &&
- Type the above code in a file and run the script in a sqlplus session
- The package body manage_students is compiled into the database.

Stored Code

- Functions in packages need to meet additional restrictions in order to be used in a SELECT statement (must be row functions and using only SQL datatypes, and have no DML(insert, update, delete), have certain level of purity achieved with PRAGMA RESTRICT_REFERENCES, p 332, 358-361, 366-368).
- Getting Stored Code Information from the Data Dictionary
- 1. DESC USER_ERRORS
  [used to determine details of a compilation error]
- 2. SHO ERR
  [displays the line number the error occurred in USER_SOURCE view]
- 3. DESC <packagename>
  To query the data dictionary to determine all stored objects in the current schema of the database including the current status of the stored code, use:

Stored Code

- SELECT OBJECT_TYPE, OBJECT_NAME, STATUS
FROM USER_OBJECTS
WHERE OBJECT_TYPE IN
  ('FUNCTION', 'PROCEDURE', 'PACKAGE',
'PACKAGE_BODY')
ORDER BY OBJECT_TYPE;
- 4. We can retrieve information from USER_ERRORS view with
  SELECT line || '/' || position "LINE/COL", TEXT "ERROR"
  FROM user_errors
  WHERE name = 'FORCE_ERROR';
- 5. DESC USER_DEPENDENCIES
  [used to analyze impact of table changes]
- 6. SELECT referenced_name
FROM user_dependencies
WHERE name = 'SCHOOL_API';
- The above lists all objects referenced in the package.

Stored Code

- 7. DEPTREE is an Oracle utility that shows which objects are dependent on a given object, but DBA access is needed to use this utility [see page 365 for details]
- 8. What is purity level of a function in a package?
  Purity level of a function describes the extent to which the function is free of side effects (altering public values also used by other functions)
- Available Purity levels are
  - WNDS (write no database state) or does not change any database tables
  - WNPS (write no package state) or does not alter any package variables
  - RNPS (reads no package state)
  - RNDS (reads no database state or table)

Stored Code

- To assert Purity Level, use
  PRAGMA RESTRICT_REFERENCES
  (function_name, WNDS[, WNPS][,RNDS][,RNPS]);
- 10. With the Purity level set as:

PRAGMA RESTRICT_REFERENCES (school_api, WNDS, WNPS);

- Inside the package specification, any update instruction will result in a purity level violation error.
- Only the WNDS level is mandatory and we need a separate pragma statement for each packaged function used in an SQL statement.
- The pragma must come after the function declaration in the package specification

Overloading Modules

- OVERLOADING MODULES
- When we overload modules, we give two or more modules the same name.
- The parameter lists of the modules should differ enough to have the versions distinguishable.
- Modules can be overloaded in the following 3 contexts.
- in a local module in the same PL/SQL block
- in a package specification
- in a package body.
- [see page 359-361]
- E.g., the following two procedures cannot be overloaded.
- PROCEDURE calc_total (reg_in IN CHAR);
- PROCEDURE calc_total (reg_in IN VARCHAR2);

Triggers

- **TRIGGERS**
- **A database trigger is a named PL/SQL block stored in a database and executed when a triggering event occurs.**
- **Executing a trigger is called firing a trigger.**
- **A triggering event is a DML (INSERT, UPDATE, or DELETE) statement executed against a database table.**
- **A trigger can fire before or after a triggering event**
- **For example, a trigger can be defined to fire before an INSERT statement on the STUDENT table and it fires each time before you insert a row in the STUDENT table.**

Triggers

- The general syntax for creating a trigger is:

```
CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE |
    AFTER}
    Triggering_event ON table-name [FOR EACH ROW]
    [WHEN condition]
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END;
```

Triggers

- Dropping a table also drops all triggers on the table.
- Triggers can be used to enforce complex business rules not handled with integrity constraints.
- Maintaining security rules
- Automatically generating values for derived columns
- Collecting statistical information on table access.
- Preventing invalid transactions
- For auditing
- A trigger may not issue a COMMIT, SAVEPOINT or ROLLBACK statement.

Triggers

- **Any function or procedure called by a trigger may not issue a transactional control statement (COMMIT, SAVEPOINT, ROLLBACK)**
- **Datatype LONG and LONG RAW cannot be used in a trigger, E.g.,**

```
CREATE OR REPLACE TRIGGER student_bi
BEFORE INSERT ON student
FOR EACH ROW
DECLARE
```

Triggers

```
v_student_id STUDENT.STUDENT_ID % TYPE;
BEGIN
    SELECT STUDENT_ID_SEQ.NEXTVAL
    INTO v_student_id
    FROM dual;
    :NEW.student_id := v_student_id;
    :NEW.created_by := USER;
    :NEW.created_date := SYSDATE;
    :NEW.modified_by := USER;
    :NEW.modified_date := SYSDATE;
END;
```

Triggers

- The above trigger fires before each INSERT statement on the student table.
- The pseudo-record :NEW accesses a row currently being processed.
- The :NEW record is a type TRIGGERING_TABLE % TYPE and in this case, it is of type STUDENT % TYPE and members (attributes) of this record are accessed using the dot notation (eg, :NEW.student_id).
- Once the above trigger is used to populate the record with student_id, user and creation dates, the attributes left to insert values in this record would be last and first names, zip and registration date.
- Thus, the shorter version of INSERT used is to accomplish this is:

Triggers

- INSERT INTO student (first_name, last_name, zip, registration_date)
- VALUES ('John', 'Smith', 'OO914', SYSDATE);
- BEFORE triggers should be used
- When the trigger provides values for derived columns before an INSERT or UPDATE statement is completed.
- When the trigger determines whether an INSERT, UPDATE or DELETE statement should be allowed to complete. (E.g., determining if an inserted ZIP is valid)

Triggers

- AFTER TRIGGERS
- Example: the statistics table with structure statistics (Table_Name, Transaction_Name, Transaction_user, Transaction_Date);
- A trigger on the Instructor table, which fires after an UPDATE or INSERT statement is:

```
CREATE OR REPLACE TRIGGER instructor_aud
    BEFORE UPDATE OR DELETE ON INSTRUCTOR
    DECLARE
   v_type VARCHAR2(10);
    BEGIN
   IF UPDATING
   THEN
     v_type := 'UPDATE';
```

Triggers

```
  ELSEIF DELETING
  THEN
    v_type := 'DELETE';
  END IF;
  UPDATE statistics
  SET transaction_user = USER
        transaction_date = SYSDATE
  WHERE table_name = 'INSTRUCTOR'
        AND  transaction_name = v_type;
  IF SQL % NOTFOUND
  THEN
    INSERT INTO statistics
    VALUES (' INSTRUCTOR', v_type, USER, SYSDATE);
  END IF;
END;
```

Triggers

- Note that the functions UPDATING and DELETING are Boolean.
- This trigger updates or inserts a record in the statistics table when an UPDATE or DELETE operation against the instructor table occurs.
- Once trigger is created on the instructor table, any UPDATE or DELETE causes modification of old record or creating of new records, in the statistics.
- After triggers should be used when
- a trigger should be fired after a DML statement is executed.

Triggers

- When a trigger performs actions not specified in a BEFORE trigger.
- Consider the following UPDATE statement.

```
UPDATE student
SET zip = '01247'
WHERE zip = '02189';
```

- The value "01247" of the ZIP column is a new value and trigger would reference it as :NEW.ZIP. The value "02189" in the ZIP column is the previous value and is referenced as :OLD.ZIP.

Triggers

- :OLD is not defined for INSERT statements and :NEW is not defined for DELETE statements.
- These pseudo variables are referenced in the condition of a WHEN statement without : as in:

```
CREATE TRIGGER student_au
BEFORE UPDATE ON STUDENT
FOR EACH ROW
WHEN (NVL(NEW.ZIP, ‘ ‘) < > OLD.ZIP)
   Trigger Body ……
```

Types of Triggers

- **TYPES OF TRIGGERS**
- **Row Triggers**
- **A row trigger is defined with a statement including FOR EACH ROW as in**

```
CREATE OR REPLACE TRIGGER course_au
AFTER UPDATE ON COURSE
FOR EACH ROW
……
```

- **A row trigger fires as many times as there are rows affected by the trigger.**
- **Statement trigger**
- **A statement trigger does not include FOR EACH ROW in its definition, E.g.,**

```
CREATE OR REPLACE TRIGGER enrollment_ad
AFTER DELETE ON ENROLLMENT
……
```

Types of Triggers

The trigger fires once after a DELETE statement is issued against the enrollment table.

- Statement triggers are used for actions that do not depend on individual records.
- INSTEAD OF TRIGGERS
- An instead of trigger is a row trigger that is defined on views to fire instead of the DML statement.

Mutating Table Issues/Trigger Restrictions

- **MUTATING TABLE ISSUES**
- **A mutating table is a table having a DML statement issued against it. For a trigger, it is the table on which this trigger is defined.**
- **A constraining table is a table read from, for a referential integrity constraint.**
- **TRIGGER SQL Statement Restrictions**
- **An SQL statement may not read or modify a mutating table.**
- **An SQL statement may not modify columns of constraining table having primary, foreign, or unique constraints defined on them.**

PL/SQL Tables

- PL/SQL Tables
- PL/SQL tables are PL/SQL arrays and DML statements cannot be issued on them.
- PL/SQL tables exist in memory only and not in database.
- Declaration of PL/SQL table
- To declare PL/SQL table,
- Define the table structure using TYPE statement.
- Declare the actual table.
- E.g., declaration of PL/SQL table

PL/SQL Tables

```
DECLARE
   TYPE LnameType IS TABLE OF
--Table structure definition
   Student.last_name % TYPE
   INDEX BY BINARY_INTEGER;
   --Create the actual table
   Slname LnameType;
   Ilname LnameType;
BEGIN
      NULL;
      ……
END;
```

PL/SQL Tables

- Referencing and Modifying PL/SQL Table Rows
- A particular table row is referenced as:
- <table_name> (<index_value>)
- The datatype of the index value is compatible with BINARY_INTEGER datatype and we assign values to a row using the := operator.
- E.g.

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR c_slname IS
    SELECT last_name, student_id, ROWNUM
    FROM student
    WHERE student_id < 110
    ORDER BY last_name;
```

PL/SQL Tables

```
TYPE type_lname_tab IS TABLE OF
        student.last_name % TYPE
        INDEX BY BINARY_INTEGER;
        tab_slname type_lname_tab;
        v_slname_counter NUMBER:=0;
  BEGIN
        FOR r_slname IN c_slname
        LOOP

         v_slname_counter := v_slname_counter + 1;

        tab_slname(v_slname_counter):=r_slname.last_name;
        END LOOP;
```

PL/SQL Tables

```
FOR i_slname IN 1..v_slname_counter
    LOOP
        DBMS_OUTPUT.PUT_LINE('Here is a last name:' || Tab_slname(i_slname));
    END LOOP;
END;
```

PL/SQL Attributes

- **PL/SQL Table Attributes**
- **Attributes used to gain information on a PL/SQL table are:**
- **1. DELETE – deletes rows in a table**
- **2. EXISTS – returns TRUE if specified entry exists in table.**
- **3. COUNT- returns number of rows in table.**
- **4. FIRST – returns the index of the first row in table.**
- **5. LAST – returns the index of the last row in table.**
- **6. NEXT – returns the index of the next row in table.**
- **7. PRIOR – returns index to previous row in table.**

PL/SQL Attributes

- Syntax of Use of Table Attributes
- PL/SQL table attributes are used with the following syntax
- <table_name>. <attribute>
- E.g., with a table name t_student, we can assign the row count of this table to variable v_count as follows:
- v_count := t_student.count;
- t_student.delete deletes all rows from the t_student table.
- t_student.delete(15) deletes only the 15th row. Also t_student.exists(100) will work on the 100th row.
- Thus, for some attributes, the syntax involes specifying which rows as:
- <table_name>.<attribute> (<index number>[, <index number>])