Notes on PL/SQL

Zannatun Naim Sristy
Lecturer
zannatunnaim@iut-dhaka.edu
Mohammad Anas Jawad
Lecturer
anasjawad@iut-dhaka.edu
Md. Bakhtiar Hasan
Assistant Professor
bakhtiarhasan@iut-dhaka.edu
Department of Computer Science and Engineering
Islamic University of Technology

CONTENTS

CHAPTER 1 INTRODUCTION

Procedural Language for Structured Query Language (PL/SQL) lets you write code once and deploy it in the database nearest to data. PL/SQL can simplify application development, optimize execution, and improve resource utilization in the database. The language is a case-insensitive programming language, like SQL.

One of the most important aspects of PL/SQL is its tight integration with SQL. You do not need to rely on any intermediate software "glue" such as ODBC (Open Database Connectivity) or JDBC (Java Database Connectivity) to run SQL statements in your PL/SQL programs. Instead, you just insert the `UPDATE` or `SELECT` into your code.

1.1 History

PL/SQL was developed by modeling concepts of structured programming, static data typing, modularity, exception management, and parallel (concurrent) processing found in the Ada programming language. The *Ada* programming language, developed for the United States Department of Defense, was designed to support military real-time and safety-critical embedded systems, such as those in airplanes and missiles. The Ada programming language borrowed significant syntax from the *Pascal* programming language, including the assignment and comparison operators and the single-quote delimiters.

1.2 Basic Structure

PL/SQL is a blocked programming language. Program units can be named or unnamed blocks. Unnamed blocks are known as *anonymous blocks* and are labeled so throughout the handout. The PL/SQL coding style differs from that of the C, C++, and Java programming languages. For example, curly braces do not delimit blocks in PL/SQL.

Anonymous-block programs are effective in some situations. You typically use anonymous blocks when building scripts to seed data or perform one-time processing activities. They are also effective when you want to nest activity in another PL/SQL block's execution section. The basic anonymous-block structure must contain an execution section. You can also put optional declaration and exception sections in anonymous blocks. The following illustrates an anonymous-block prototype:

```
[DECLARE]
    declaration_statements
BEGIN
    execution_statements
[EXCEPTION]
    exception_handling_statements
END;
/
```

1.2.1 Declaration Block

The declaration block lets you define datatypes, structures, and variables. Defining a variable means that you give it a name and a datatype. You can also declare a variable by giving it a name, a datatype, and a value. You both define and assign a value when declaring a variable.

Structures and cursors can be defined here as well. Structures are compound variables, like collections, record structures, or system reference cursors. Structures can also be locally named functions, procedures, or cursors. Cursors act like little functions. Cursors have names, signatures and a return type - the output columns from a query or `SELECT` statement. The `DECLARE` reserved word begins the declaration block, and the `BEGIN` reserved word ends it.

1.2.2 Execution Block

The execution block lets you process data. It can contain variable assignments, comparisons, conditional operations, and iterations. Also, the execution block is where you access cursors and other named program units. Functions, procedures, and some object types are named program units. You can also nest anonymous-block programs inside the execution block. The BEGIN reserved word starts the exception block, and the optional `EXCEPTION` or required `END` reserved word ends it. You must have at least one statement inside an execution block.

1.2.3 Exception Block

The exception handling block lets you manage exceptions. You can both catch and manage them there. The exception block allows for alternative processing; in many ways it acts like combination of a catch and finally block in the Java programming language. The `EXCEPTION` reserved word starts the section, and the `END` reserved word ends it.

1.3 Example Programs

Let's write our first minimal block that will do nothing:

```
BEGIN
    NULL;
END;
/
```

This block does nothing except let the compilation phase complete without an error. Compilation, like other languages, requires syntax parsing. The lack of a statement in the block raises a parsing error. Every PL/SQL block must contain something, at least a `NULL;` statement, or it will fail runtime compilation.

1.3.1 Output

Hers' an example of a “Hello, World!” program:

```
SET SERVEROUTPUT ON SIZE 1000000

BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, World!');
END;
/
```

Output:

```
Hello, World!
```

To print content on the console, we need to enable the SQL*Plus `SERVEROUTPUT` variable. This is because whatever we output using the keyword `DBMS_OUTPUT.PUT_LINE` is internally stored inside a buffer in *Shared Global Area* (SGA) memory area of size 2000 bytes. Here, `SIZE` sets the size of the buffer. In order to fetch it from that buffer, we need to set the environment variable for the session. Note that, we need to set the environment variable only once for a session.

1.3.2 Input

SQL*Plus supports the use of substitution variables in the interactive console, which are prefaced by an ampersand (&). Substitution variables are variable-length strings or numbers, using which we can take input from the console:

```
DECLARE
    USERNAME VARCHAR2(5);
BEGIN
    USERNAME := '&username';
    DBMS_OUTPUT.PUT_LINE('Hello ' || USERNAME);
END;
/
```

Output:

```
Enter value for username: Alice
old   4:     USERNAME := '&username';
new   4:     USERNAME := 'Alice';
Hello Alice
```

The line starting with `old` designates where your program accepts a substitution, and `new` designates the run-time substitution. You can suppress echoing the substitution by using `SET VERIFY OFF`.

1.3.3 Error Handling

In the last program, assigning a string literal that is too large for the variable fires an exception. Exception blocks manage raised errors. A generic exception handler manages any raised error. You use a `WHEN` block to catch every raised exception with the generic error handler - `OTHERS`.

```
DECLARE
    USERNAME VARCHAR2(5);
BEGIN
    USERNAME := '&username';
    DBMS_OUTPUT.PUT_LINE('Hello ' || USERNAME);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

Output:

```
Enter value for username: Charlie
ORA-06502: PL/SQL: numeric or value error: character string buffer too
small
```

CHAPTER 2 VARIABLES, OPERATORS, AND DATATYPES

2.1 Variables

Variable names begin with letters and can contain alphabetical characters, ordinal numbers (0 to 9), the $, _, and # symbols. Similar to SQL, the names are case-insensitive. Variables have local scope only, meaning they are available only in the scope of a given PL/SQL block. The exception to that rule are nested anonymous blocks. Nested anonymous blocks operate inside the defining block. They can thereby access variables from the containing block. That is, unless you have declared the same variable name as something else inside the nested anonymous block.

A declaration of a number variable without an explicit assignment makes its initial value null. The prototype shows that you can assign a value later in the execution block:

```
DECLARE
    VARIABLE_NAME NUMBER;
BEGIN
    VARIABLE_NAME := 1;
END;
/
```

An explicit assignment declares a variable with a not-null value. You can use the default value or assign a new value in the execution block. Both are demonstrated below:

```
DECLARE
    VARIABLE_NAME NUMBER [:= | DEFAULT] 1;
BEGIN
    VARIABLE_NAME := 1;
END;
/
```

A list of reserved words and keywords, which should not be used as identifier (e.g., variables, functions, cursors, etc.) names are provided in “`SYS.V$RESERVED_WORDS`” view.

2.2 Operators

Some common operators are as follows:

Table 2.1: A list of common operators in PL/SQL

Symbol	Type	Description
+	Arithmetic	The addition operator lets you add left and right operands and returns a result.

Symbol	Type	Description
`-`	Arithmetic	The subtraction operator lets you subtract the right operand from the left operand and returns a result.
`*`	Arithmetic	The multiplication operator lets you multiply a left operand by a right operand and returns a result.
`/`	Arithmetic	The division operator lets you divide a left operand by a right operand and returns a result.
`**`	Arithmetic	The exponential operator raises a left operand to the power designated by a right operand. The operator enjoys the highest precedence for math operators in the language. As a result of that, a fractional exponent must be enclosed in parentheses (also known as expression or list delimiters) to designate order of operation. Without parentheses, the left operand is raised to the power of the numerator and the result divided by the denominator of a fractional exponent.
`:=`	Assignment	The assignment operator is a colon immediately followed by an equal symbol. It is the only assignment operator in the language. Remember that the left operand for this operator must always be a variable. The right operand an be a value, variable, or function. Functions must return a value when they are right operands. This is convenient in PL/SQL because all functions return values. Functions in this context are also known as expressions
`=`	Relational	The equal symbol is the comparison operator. It tests for equality of value and implicitly does type conversion where possible. There is no identity comparison operator because PL/SQL is a strongly typed language. PL/SQL comparison operations are equivalent to identity comparisons because you can only compare like typed values.
`!=`	Relational	There are three not-equal comparison operators: `!=`, `^=`, and `<>`. They all perform exactly the same behaviors. You can use whichever suits your organizational needs.
`>`	Relational	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
`<`	Relational	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
`>=`	Relational	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
`<=`	Relational	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
`LIKE`	Comparison	Compares a character, string, or `CLOB` value to a pattern and returns true if the value matches the pattern and false if it does not.
`BETWEEN`	Comparison	Tests whether a value lies in a specified range.
`IN`	Comparison	Tests set membership.
`IS NULL`	Comparison	Returns true if its operand is null, or false otherwise. Comparisons involving null values always yield null.

Symbol	Type	Description
`IS EMPTY`	Comparison	Allows you to check whether a `VARRAY` collection variable is empty. Empty means that the collection was constructed without any default elements.
`MEMBER OF`	Comparison	Lets you find out whether an element is a member of a collection. Only works with collections of scalar SQL datatypes.
`AND`	Logical	If both the operands are true, then condition becomes true. It also allows you to combine two comparisons into one.
`OR`	Logical	If any of the two operands are true, then condition becomes true.
`NOT`	Logical	Used to reverse the logical state of its operand.
`.`	Association	The component selector is a period, and it glues references together. It is often used to link a schema and a table, a package and a function, or an object and a member method. Component selectors are also used to link cursors and cursor attributes (columns). For example, `INSTRUCTOR.SALARY.`
`@`	Association	The remote access indicator lets you access a remote database through database links. It can also be used to run scripts.
`:`	Association	The host variable indicator precedes a valid identifier name, and designates that identifier as a session-level variable. Session-level variables are also known as *bind variables*. You use SQL*Plus to define a session-level variable. Only the `CHAR`, `CLOB`, `NCHAR`, `NCLOB`, `NUMBER`, `NVARCHAR2`, `REFCURSOR`, and `VARCHAR2` datatypes are available for session variables.
`&`	Association	The substitution indicator lets you pass actual parameters into anonymous-block PL/SQL programs. You should never assign substitution variables inside declaration blocks because assignment errors do not raise an error that you can catch in your exception block. You should make substitution variable assignments in the execution block.
`%`	Association	The attribute indicator lets you link a database catalog column, row, or cursor attribute. You are anchoring a variable datatype when you link a variable to a catalog object, like a table or column.
`=>`	Association	The association operator is a combination of an equal sign and a greater-than symbol. It is used in name notation function and procedure calls.
`||`	Concatenation	The concatenation operator is formed by combining two perpendicular vertical lines. You use it to glue strings together.

Symbol	Type	Description
( )	Delimiter	The list delimiter is used to place a list of comma-delimited numeric or string literals, or identifiers, inside a set of parentheses. You use parentheses to enclose formal and actual parameters to subroutines or to produce lists for comparative evaluations. You can also override order of precedence by enclosing operations in parentheses. Enclosing operations in parentheses lets you override the natural order of precedence in the language.
,	Delimiter	The item separator is a comma and delimits items in lists.
--	Delimiter	In-line comment
/* */	Delimiter	Multi-line comment.
'	Delimiter	The character string delimiter is a single quote. It lets you define a string literal value.
''	Delimiter	The quoted identifier delimiter is a double quote. It lets you access tables created in case-sensitive fashion from the database catalog. This is required when you have created database catalog objects in case-sensitive fashion.
;	Statement	The statement terminator is a semicolon. You must close any statement or block unit with a statement terminator.

2.3 Datatypes

Datatypes in PL/SQL include all SQL datatypes and subtypes. PL/SQL also supports scalar and composite variables. Oracle Database 11g supports character, numeric, timestamp, binary, and row address datatypes. These are also known as SQL datatypes or built-in types because they can be used to define columns in tables and parameter datatypes in PL/SQL.

2.3.1 Scalar Datatypes

Scalar datatypes use the following prototype inside the declaration block of your programs:

```
variable_name DATATYPE [NOT NULL] [:= literal_value];
```

Boolean

The BOOLEAN datatype has three possible values: TRUE, FALSE, and NULL.

```
var1 BOOLEAN;                  -- Implicitly assigned a NULL value
var2 BOOLEAN NOT NULL := TRUE;  -- Explicitly assigned a TRUE value
var3 BOOLEAN NOT NULL := FALSE; -- Explicitly assigned a FALSE value
```

Characters and Strings

The CHAR datatype is used for fixed-length strings. On the other hand, the VARCHAR2 datatype stores variable-length character strings. For each row, Oracle Database stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle Database returns an error. Using VARCHAR2 and VARCHAR saves on space used by the table.

Remember, the maximum size of both CHAR and VARCHAR2 in PL/SQL is greater than that of SQL. SQL allows for 4000 bytes for CHAR and VARCHAR2, whereas PL/SQL allows up to 32767 bytes. Oracle recommends you use the CLOB datatype to store character strings larger than 4000 bytes.

Difference between CHAR and VARCHAR2: The following program illustrates the memory allocation differences between CHAR and VARCHAR2 datatypes:

```
DECLARE
    C CHAR(32767) := 'hello';
    VC VARCHAR2(32767) := 'hello';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Length of C is ' || LENGTH(c));
    DBMS_OUTPUT.PUT_LINE('Length of VC is ' || LENGTH(VC));
    VC := VC || ' ';
    DBMS_OUTPUT.PUT_LINE('New length of VC is ' || LENGTH(VC));
END;
/
```

Output:

```
Length of c is 32767
Length of vc is 5
New length of vc is 6
```

The output shows that a CHAR variable sets the allocated memory size when defined. The allocated memory can exceed what is required to manage the value in the variable. The output also shows that the VARCHAR2 variable dynamically allocates only the required memory to host its value.
Difference between VARCHAR and VARCHAR2: Both VARCHAR and VARCHAR2 are used to store variable-length character strings. Here are a few differences between them:

Table 2.2: Difference between VARCHAR and VARCHAR2

VARCHAR	VARCHAR2
Can differentiate between NULL and empty string.	Considers both as the same.
ANSI SQL Standard.	Oracle Standard.
Definition may change in future	Definition will not change.

More on VARCHAR2: Globalization support allows the use of various character sets for the character datatypes. It lets you process single-byte and multibyte character data and convert between character sets. Client sessions can use client character sets that are different from the database character set. The length semantics of character datatypes can be measured in bytes or characters.

- **Byte Semantics** treats strings as a sequence of bytes. This is the default for character datatypes.
- **Character Semantics** treat strings as a sequence of characters. A character is technically a codepoint of the database character set.

For single byte character sets, columns defined in character semantics are basically the same as those defined in byte semantics. Character semantics are useful for defining varying-width multibyte strings; it reduces the complexity when defining the actual length requirements for data storage. For example, in a Unicode database (UTF8), you might define a VARCHAR2 column that can store up to five Chinese characters together with five English characters. In byte semantics, this would require $(5 \times 3 \text{ bytes}) + (1 \times 5 \text{ bytes}) = 20$ bytes. In character semantics, the column would require 10 characters.

Consider the following example:

```
var1 VARCHAR2(100);       -- Implicitly sized at 100 bytes
var2 VARCHAR2(100 BYTE); -- Explicitly sized at 100 bytes
var3 VARCHAR2(100 CHAR); -- Explicitly sized at 100 characters
```

When we use character space allocation, the maximum size changes depending on the character set of our database. Some character sets use two or three bytes to store characters. We divide the allocated space (let: 32,767) by the number of bytes required per character, which means the maximum character limit for a VARCHAR2 is 16,383 for a two-byte character set and 10,922 for a three-byte character set.

Globalization raises a host of issues with how you use variable-length strings. You should consider using NVARCHAR2 datatypes when managing multiple character sets or Unicode.

Dates and Times

The DATE datatype is the base type for dates, times, and intervals. It contains the actual timestamp of activity. The valid range is any date from January 1, 4712 BCE to December 31, 9999 CE. The most common way to capture a timestamp is to assign the SYSDATE or SYSTIMESTAMP built-in function. They both return fully qualified dates and contain all field elements of a DATE variable or column.

```
var1 DATE;                  -- Implicitly assigns a null value
var2 DATE := SYSDATE;       -- Explicitly assigns current server timestamp
var3 DATE := SYSDATE + 1;   -- Explicitly assigns tomorrow server timestamp
var4 DATE := '29-FEB-08'    -- Explicitly assigns leap year day for 2008
```

The TO_DATE() function can convert non-conforming date formats into valid DATE values. Alternatively, the CAST() function also works with the default format mask. The default format masks for dates are DD-MON-RR or DD-MON-YYYY.

You can use TRUNC(date_variable) to extract a date from a timestamp. This is useful when you want to find all transactions that occurred on a particular day. By default, the TRUNC() built-in function shaves off the time, making a date with 00 hours, 00 minutes, and 00 seconds.

```
DECLARE
    D DATE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(TRUNC(D), 'DD-MON-YY HH24:MI:SS'));
END;
/
```

Output:

```
30-OCT-22 00:00:00
```

The EXTRACT() built-in function also lets you capture the numeric month, year, or day from a DATE value.

Numbers

There are four principal number datatypes: BINARY_INTENGER, BINARY_DOUBLE/BINARY_FLOAT, NUMBER, and PLS_INTEGER. We focus on the NUMBER datatype.

The NUMBER datatype uses a custom library provided as part of the Oracle 11g database. It can store numbers in the range of 1.0×10^{-130} to 1.0×10^{126}. It does not raise a NaN (not a number) or infinity error when a literal or computational value is outside the datatype range. The following exceptions occur:

- A literal value below the minimum range value stores a zero
- A literal value above the maximum range raises a compilation error
- A computational outcome above the maximum range raises a compilation error

The NUMBER datatype supports fixed-point and floating-point numbers. Fixed-point numbers are defined by specifying the number of digits (known as the precision) and the number of digits to the right of the decimal point (known as the scale). The decimal point is not physically stored in the variable because it is calculated by the relationship between the precision and the scale. The default precision of a number is 38.

```
var1 NUMBER;          -- A null number with 38 digits
var2 NUMBER(15);      -- A null number with 15 digits
var3 NUMBER(15, 2);   -- A null number with 15 digits and 2 decimals
```

Large Objects (LOBs)

Large objects provide us with four datatypes: `BFILE`, `BLOB`, `CLOB`, and `NCLOB`. `BFILE` is a datatype that points to an external file, which limits its maximum size to 4 gigabytes. The `BLOB`, `CLOB`, and `NCLOB` are internally managed types, and their maximum size is 8 to 128 terabytes, depending on the `DB_BLOCK_SIZE` parameter value.

LOB columns contain a locator that points to where the actual data is stored. You must access a LOB value in the scope of a transaction. You essentially use the locator as a route to read data from or write data to the LOB column.

2.3.2 Composite Datatypes

There are two composite generalized datatypes: records and collections.

Record

A record, also known as a structure, typically contains a collection of related elements like a normalized database table.

```
DECLARE
    TYPE COORD2D IS RECORD
    (
        X NUMBER DEFAULT 0,
        Y NUMBER DEFAULT 0
    );
    ORIGIN COORD2D;
BEGIN
    DBMS_OUTPUT.PUT_LINE('(' || ORIGIN.X || ', ' || ORIGIN.Y || ')');
END;
/
```

Output:

```
(0, 0)
```

The execution block prints the contents of the record by using dot notation.

It is possible to nest records. You access the names of the nested records by using another component selector.

```
DECLARE
    TYPE FULL_NAME IS RECORD
    (
        FIRST VARCHAR2(10 CHAR) := 'John',
        LAST VARCHAR2(10 CHAR) := 'Doe'
    );
    TYPE employee IS RECORD
    (
        ID NUMBER DEFAULT 1,
        NAME FULL_NAME
    );
    E1 EMPLOYEE;
BEGIN
    DBMS_OUTPUT.PUT_LINE(E1.ID);
    DBMS_OUTPUT.PUT_LINE(E1.NAME.FIRST || ' ' || E1.NAME.LAST);
END;
/
```

Output:

```
1
John Doe
```

Records are extremely useful when working with cursors and collections. Records are exclusively available inside your PL/SQL execution scope. You can define a stored function to returning a record type, but that limits how you can use the function. SQL can only access stored functions when they return SQL datatypes.

Collection

Collections are sets of like things. The things can be scalar variables, large objects, user-defined objects, or records.

```
DECLARE
    TYPE NUMBER_ARRAY IS VARRAY(10) OF NUMBER;
    LIST NUMBER_ARRAY := NUMBER_ARRAY(1, 2, 3, 4, 5, 6, 7, 8, NULL, NULL)
   ;
BEGIN
    FOR i in 1..LIST.LIMIT LOOP
        DBMS_OUTPUT.PUT('[' || LIST(i) || ']');
    END LOOP;
    DBMS_OUTPUT.NEW_LINE;
END;
/
```

Output:

```
[1][2][3][4][5][6][7][8][][]
```

The program initializes the first eight elements with values and last two with nulls. The declaration allocates space for ten elements by setting all elements to a value, which can include a null. The LIMIT() method returns the maximum size. The LOOP statement is self-explanatory, but we will learn more about it later.

2.3.3 Datatype Indicator

We can declare a variable to be the same type as an attribute of a table. A variable can store the entire row of a table or a single column. For example, consider the following table:

```
CREATE TABLE EMPLOYEE
(
    ID NUMBER(5) NOT NULL,
    DEPT_ID VARCHAR2(20),
    DEPT_NAME VARCHAR2(20) DEFAULT 'HR',
    CONSTRAINT DEPT_ID_RANGE CHECK (DEPT_ID BETWEEN 10 AND 20)
);
```

We can use the datatype mentioned in the table in PL/SQL directly:

```
DECLARE
    EMPREC EMPLOYEE%ROWTYPE;
    ID EMPLOYEE.ID%TYPE;
BEGIN
    EMPREC.ID := NULL;     -- Constraint not inherited
    ID := 1000000002;      -- Invalid, number precision too large
    EMPREC.DEPT_ID := 50; -- Constraint not inherited
    /* Default value not inherited */
    DBMS_OUTPUT.PUT_LINE('Employee dept name: ' || EMPREC.DEPT_NAME);
END;
/
```

Output:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 6
```

As demonstrated by the example, constraints on datatypes are inherited. However, other constraints are not inherited. Even if we set the precision of the right operand of `ID` variable assignment to less than 6, we get:

```
Employee dept name:
```

That means, default values are not inherited as well.

CHAPTER 3 CONTROL STRUCTURES

Control structures let you make conditional choices, repeat operations, and access data. The IF and CASE statements let you branch program execution according to one or more conditions. Loop statements let you repeat behavior until conditions are met. Cursors let you access data one row or one set of rows at a time.

3.1 Conditional Statements

Conditional statements check whether a value meets a condition before taking action. There are two types of conditional structures in PL/SQL. One is the IF statement, and the other is the CASE statement. The IF statement that has two subtypes, *if-then-else* and *if-then-elsif-then-else*. The *elsif* is not a typo but the correct reserved word in PL/SQL. This is another legacy from Pascal and ADA.

3.1.1 IF Statement

IF statements evaluate a condition. The condition can be any comparison expression, or set of comparison expressions that evaluates to a logical true or false. You can use comparison operators and symbols like: AND, BETWEEN, IN, IS EMPTY, IS NULL, IS A SET, LIKE, MEMBER OF, NOT, OR, etc.

All IF statements are blocks in PL/SQL and end with the END IF phrase.

```
DECLARE
  X NUMBER ;
BEGIN
  X := 10;
  IF (X = 0) THEN
    DBMS_OUTPUT.PUT_LINE('The value of x is 0');
  ELSIF (X BETWEEN 1 AND 10 ) THEN
    DBMS_OUTPUT.PUT_LINE('The value of x is between 1 and 10');
  ELSE
    DBMS_OUTPUT.PUT_LINE('The value of x is greater than 10');
  END IF;
END;
/
```

Output:

```
The value of x is between 1 and 10
```

One possible use-case of the IF statements can be categorizing different students:

```
DECLARE
    CGPA NUMBER;
    STUDENT_ID NUMBER := 170042043;
```

```
BEGIN
    SELECT MAX(CGPA) INTO CGPA
        FROM STUDENTS
            WHERE ID = STUDENT_ID;
    IF (CGPA > 3.8) THEN
        DBMS_OUTPUT.PUT_LINE('Brilliant');
    ELSIF (CGPA BETWEEN 3.5 AND 3.8) THEN
        DBMS_OUTPUT.PUT_LINE('Mid Level');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Poor');
    END IF;
END;
/
```

Output:

```
Brilliant
```

3.1.2 Simple CASE Statement

The simple CASE statement sets a selector. A selector is a variable, function, or expression that the CASE statement attempts to match in WHEN blocks. The selector immediately follows the reserved word CASE. You can use any PL/SQL datatype except a BLOB, BFILE, or composite types.

```
DECLARE
    SELECTOR NUMBER := 1;
BEGIN
    CASE SELECTOR
        WHEN 0 THEN
            DBMS_OUTPUT.PUT_LINE('Case 0!');
        WHEN 1 THEN
            DBMS_OUTPUT.PUT_LINE('Case 1!');
        ELSE
            DBMS_OUTPUT.PUT_LINE('No Match!');
    END CASE;
END;
/
```

Output:

```
Case 1!
```

3.1.3 Searched CASE Statements

The selector is implicitly set for a search CASE statement. For example, if we consider the university schema provided in the 'Silberschatz, Korth, Sudarshan - Database System Concepts (7th Edition)' book, the following program can be used to categorize the instructors based on their salaries:

```
SELECT NAME, ID, (CASE TRUE
                WHEN SALARY < 50000 THEN 'Low'
                WHEN SALARY BETWEEN 50000 AND 70000 THEN 'Medium'
                WHEN SALARY > 70000 THEN 'High'
                ELSE 'N/A'
                END) SALARY
    FROM INSTRUCTOR
        ORDER BY NAME;
```

Output:

```
NAME                 ID    SALARY
-------------------- ----- ------
Brandt               83821 High
Califieri            58583 Medium
Crick                76766 High
Einstein             22222 High
El Said              32343 Medium
Gold                 33456 High
Katz                 45565 High
Kim                  98345 High
Mozart               15151 Low
Singh                76543 High
Srinivasan           10101 Medium
Wu                   12121 High
```

You can omit the TRUE because it is the default selector, but it is recommended that you do not. Putting it in adds clarity.

3.2 Iterative Structures

Iterative statements are blocks that let you repeat a statement or set of statements. There are two types of iterative statements. One guards entry into the loop before running repeatable statements. The other guards exit. An iterative statement that only guards exit guarantees that its code block is always run once and is known as a repeat until loop block.

PL/SQL supports FOR, SIMPLE, and WHILE loops. Loops typically work in conjunction with cursors, but can work to solve other problems, like searching or managing Oracle collections.

3.2.1 LOOP and EXIT Statements

Simple loops are explicit block structures. A simple loop starts and ends with the LOOP reserved word. An EXIT statement or an EXIT WHEN statement is required to break the loop.

If we use conditional statements:

```
DECLARE
    X NUMBER := 10;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE(X);
        X := X + 10;
        IF X > 50 THEN
            EXIT;
        END IF;
    END LOOP;
    -- Control resumes here, after EXIT
    DBMS_OUTPUT.PUT_LINE('After EXIT, X is: ' || X);
END;
/
```

Output:

```
10
20
30
40
50
After EXIT, X is: 60
```

If we use conditional `EXIT WHEN`:

```
DECLARE
    X NUMBER := 10;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE(X);
        X := X + 10;
        EXIT WHEN X > 50;
    END LOOP;
    -- Control resumes here, after EXIT
    DBMS_OUTPUT.PUT_LINE('After EXIT, X is: ' || X);
END;
/
```

Output:

```
10
20
30
40
50
After EXIT, X is: 60
```

Skipping an index value is also possible in Oracle 11g by using the new `CONTINUE` statement. It signals an immediate end to a loop iteration and returns tot he first statement in the loop.

3.2.2 FOR Loop

A `FOR .. LOOP` is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. After the body of the for loop executes, the value of the counter variable is increased or decreased. The initial value and the final value of the loop variable or counter can be literals, variables, or expressions, but they must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception `VALUE_ERROR`. The initial value need not be 1; however, the loop counter increment (or decrement) must be 1. PL/SQL allows to determine the loop range dynamically at run-time.

```
DECLARE
    i NUMBER(2);
BEGIN
    FOR i IN 10 .. 20 LOOP
        DBMS_OUTPUT.PUT_LINE('Value of i: ' || i);
    END LOOP;
END;
/
```

Output:

```
Value of i: 10
Value of i: 11
Value of i: 12
Value of i: 13
Value of i: 14
Value of i: 15
Value of i: 16
Value of i: 17
Value of i: 18
Value of i: 19
Value of i: 20
```

There is no exit statement in the example because one is not required. The exit statement is implicitly placed at the top of the loop. The conditional logic checks whether the range index is less than the top of the range, and it exits when that condition is not met.

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the upper bound. You can reverse this order by using the REVERSE keyword. After each iteration, the loop counter is decremented. Remember that, you must write the range bounds in ascending order.

```
DECLARE
    i NUMBER(2);
BEGIN
    FOR i IN REVERSE 10 .. 20 LOOP
        DBMS_OUTPUT.PUT_LINE('Value of i: ' || i);
    END LOOP;
END;
/
```

Output:

```
Value of i: 20
Value of i: 19
Value of i: 18
Value of i: 17
Value of i: 16
Value of i: 15
Value of i: 14
Value of i: 13
Value of i: 12
Value of i: 11
Value of i: 10
```

3.2.3 WHILE Loop

The WHILE loop differs from the simple loop because it guards entry to the loop, not exit. It sets the entry guard as a precondition expression. The loop is only entered when the guard condition is met. For example, the following program can be used to determine the prime factors of a number:

```
DECLARE
    N NUMBER := 10;
    DIV NUMBER := 2;
BEGIN
    DBMS_OUTPUT.PUT_LINE('The prime factors of ' || N || ' are:');
    WHILE (N > 1) LOOP
        IF (MOD(N, DIV) = 0) THEN
            DBMS_OUTPUT.PUT_LINE(DIV);
            N := TRUNC(N / DIV);
        ELSE
            DIV := DIV + 1;
        END IF;
    END LOOP;
END;
/
```

Output:

```
The prime factors of 10 are:
2
5
```

Here, `MOD` function is used to determine the modulo and `TRUNC` function is used to extract the integer portion of the quotient.

CHAPTER 4 FUNCTIONS AND PROCEDURES

Functions and Procedures are two of the key components of PL/SQL stored programming units. Oracle maintains a unique list of stored object names for tables, views, sequences, stored programs, and types. This list is known as a namespace. Functions and procedures are in this namespace.

Stored functions and procedures provide a way to hide implementation details in a program unit. They also let you wrap the implementation from prying eyes on the server tier. However, the main motivation for Functions and Procedures is modular code. Modularization is the process by which you break up large blocks of code into smaller pieces (modules) that can be called by other modules. Modularization of code is analogous to normalization of data, with many of the same benefits and a few additional advantages. With modularization, your code becomes:

- **Reusable:** By breaking up a large program or entire application into individual components that plug-and-play together, you will usually find that many modules are used by more than one other program in your current application. Designed properly, these utility programs could even be of use in other applications!

- **Manageable:** Which would you rather debug: a 1,000-line program or five individual 200-line programs that call each other as needed? Our minds work better when we can focus on smaller tasks. You can also test and debug on a per program scale (called unit testing) before individual modules are combined for a more complicated integration test.

- **Readable:** Modules have names, and names describe behavior. The more you move or hide your code behind a programmatic interface, the easier it is to read and understand what that program is doing. Modularization helps you focus on the big picture rather than on the individual executable statements. You might even end up with that most elusive kind of software: self-documenting code.

- **Reliable:** The code you produce will have fewer errors. The errors you do find will be easier to fix because they will be isolated within a module. In addition, your code will be easier to maintain because there is less of it and it is more readable.

4.1 Procedure

A procedure is a module that performs one or more actions. Because a procedure call is a standalone executable statement in PL/SQL, a PL/SQL block could consist of nothing more than a single call to a procedure. Procedures are key building blocks of modular code, allowing you to both consolidate and reuse your program logic. The following illustrates a procedure prototype:

```
[CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
```

```
[AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
[PRAGMA AUTONOMOUS_TRANSACTION;]
    [local_declarations]
BEGIN
    executable_statements
[EXCEPTION
    exception_handlers]
END [procedure_name];
```

Procedures cannot be right operands. Nor can you use them in SQL statements. You move data into and out of PL/SQL stored procedures through their formal parameter list. Here, parameter modes define the behavior of formal parameters. These parameter modes offer you the ability to use pass-by-value or pass-by-reference formal parameters. The three parameter modes: `IN` (default), `OUT`, and `IN OUT` can be used with any subprogram:

- `IN`: The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter acts like a PL/SQL constant. It is considered read-only, i.e., it cannot be changed.
- `OUT`: Any value the actual parameter has when the procedure is called is ignored. Inside the procedure, the formal parameter acts like an uninitialized PL/SQL variable and thus has a value of `NULL`. It can be read from and written to.
- `IN OUT`: This mode is a combination of `IN` and `OUT`.

During the declaration of the parameters in Procedures, you must leave out the constraining part of the declaration.

The following procedure can be used to determine the salary of an instructor given their ID:

```
-- Create procedure
CREATE OR REPLACE
PROCEDURE FIND_SAL(I_ID IN NUMBER, SALARY OUT NUMBER)
AS
BEGIN
    SELECT MAX(SALARY) INTO SALARY
        FROM INSTRUCTOR
            WHERE ID = I_ID;
END;
/

-- Call it from an anonymous block
DECLARE
    AMOUNT NUMBER;
BEGIN
    FIND_SAL(10101, AMOUNT);
    DBMS_OUTPUT.PUT_LINE(AMOUNT);
END;
/
```

Output:

```
65000
```

4.2 Function

A function is a module that returns data through its `RETURN` clause, rather than in an `OUT` or `IN OUT` parameter. Unlike a procedure call, which is a standalone executable statement, a call to a function can exist only as part of an executable statement, such as an element in an expression or the value assigned as the default in a declaration of a variable. Functions are convenient structures because you can call them directly from SQL statements or PL/SQL programs. They can also be used as right operands because they return a value. Since Functions return explicit values, it is recommended to not use OUT and IN OUT modes with functions. The following illustrates a function prototype:

```
[CREATE [OR REPLACE]]
FUNCTION function_name[(parameter[, parameter]...)]
RETURN return_type
[AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
[PRAGMA AUTONOMOUS_TRANSACTION;]
    [local_declarations]
BEGIN
    executable_statements
[EXCEPTION
    exception_handlers]
RETURN statement;
END [function_name];
```

Both procedures and functions have a name, can take parameters, return values, and be called by many users. They are stored in data dictionary. The difference is that function must return a value, but in a procedure it is optional. Also you cannot call procedures from an SQL statement.

The following program can be used to calculate the compound interest of a loan:

```
-- Declare function
CREATE OR REPLACE
FUNCTION COMPOUND_INTEREST(PA NUMBER, AIR NUMBER, TF NUMBER)
RETURN NUMBER
IS
    CI NUMBER;
BEGIN
    CI := PA * ((1 + (AIR / 100)) ** TF) - PA;
RETURN CI;
END;
/

-- Call it from an anonymous block
BEGIN
  DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(10000, 5, 5));
END;
/
```

Output:

```
2762.815625
```

Here, PA denotes the principle amount, AIR denotes the annual interest rate, TF denotes the compound period (years), and CI denotes the compound interest.

4.3 Parameter Notations

4.3.1 Positional Notation

You use positional notation to call the functions as follows:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(10000, 5, 5));
END;
/
```

4.3.2 Named Notation

You call a function using named notation by:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(TF => 5, PA => 10000, AIR => 5
   ));
END;
/
```

4.3.3 Mixed Noration

You call the function by a mix of both positional and named notation by:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(10000, TF => 5, AIR => 5));
END;
/
```

There is a restriction on mixed notation. All positional notation of actual parameters must occur first and in the same order as they are defined by the function signature.

4.3.4 Exclusionary Notation

If the formal parameters are defined as optional, you can exclude one or more of the actual parameters. For example, consider the following function:

```
CREATE OR REPLACE
FUNCTION ADD_THREE_NUMBERS
(A NUMBER := 0, B NUMBER := 0, C NUMBER := 0)
RETURN NUMBER
IS
    SUM NUMBER;
BEGIN
    SUM := A + B + C;
RETURN SUM;
END;
/
```

Here, all 3 parameters are optional. So we can write programs like:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(ADD_THREE_NUMBERS(3, C => 4));
END;
/
```

7

When you opt to not provide an actual parameter, it acts as if you are passing a null value. This is known as exclusionary notation. Oracle recommends that you should list optional parameters last in function and procedure signatures. They also recommend that you sequence optional variables so that you never have to skip an optional parameter in the list. These recommendations exist to circumvent errors when making positional notation calls.

You cannot really skip an optional parameter in a positional notation call. This is true because all positional calls are in sequence by datatype, but you can provide a comma-delimited null value when you want to skip an optional parameter in the list. However, Oracle 11g now lets you use mixed notation calls. You can now use positional notation for your list of mandatory parameters, and named notation for optional parameters. This lets you skip optional parameters without naming all parameters explicitly.

4.4 Error Handling

When creating a procedure/function, you might face compilation errors. For example, if you execute the following code:

```
/*
 ERRONEOUS CODE.
 PROCEED WITH CAUTION.
 YOU HAVE BEEN WARNED.
*/
CREATE OR REPLACE
PROCEDURE FIND_SAL(I_ID IN NUMBER, SALARY OUT NUMBER)
AS
BEGIN
    SELECT MAX(SALARY) INTO SALARY
        FROM INSTRUCTOR
            WHERE ID = I_ID
END;
/
```

Output:

```
Warning: Procedure created with compilation errors.
```

Very helpful(!) You can use `SHOW ERROR` command (or `SHO ERR` for short) right after the warning message to find out the errors:

```
SHOW ERROR
```

Output:

```
Errors for PROCEDURE FIND_SAL:

LINE/COL ERROR
-------- -----------------------------------------------------------------
4/5      PL/SQL: SQL Statement ignored
6/29     PL/SQL: ORA-00933: SQL command not properly ended
7/4      PLS-00103: Encountered the symbol "end-of-file" when expecting
         one of the following:
         ( begin case declare end exception exit for goto if loop mod
         null pragma raise return select update while with
         <an identifier> <a double-quoted delimited-identifier>
         <a bind variable> << continue close current delete fetch lock
         insert open rollback savepoint set sql execute commit forall
         merge pipe purge
```

This should help you identify that you missed a semicolon (;) in line 7.

CHAPTER 5 CURSOR

Cursor structures are the return results from SQL SELECT statements. In response to any DML statement the database creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursor
- Explicit cursor

5.1 Implicit Cursor

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit curosrs and the information in it.

Whenever a DML statement is issued, an implicit cursor is associated with this statement. For `INSERT` operations, the cursor holds the data that needs to be inserted. For `UPDATE` and `DELETE` operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like:

Table 5.1: Implicit Cursors

Attribute	Description
`%FOUND`	Returns true if an `INSERT`, `UPDATE`, or `DELETE` statement affected one or more rows, or a `SELECT INTO` statement returned one or more rows. Otherwise, it returns false.
`%NOTFOUND`	The logical opposite of `%FOUND`. It returns true if an `INSERT`, `UPDATE`, or `DELETE` statement affected no rows, or a `SELECT INTO` statement returned no rows.

Attribute	Description
%ISOPEN	Always returns false for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	This attributes returns the number of rows changed by an INSERT, UPDATE, or DELETE statement or the number of rows returned by a SELECT INTO statement.

For example, the following program counts the number of rows affected by the UPDATE statement:

```
DECLARE
    TOTAL_ROWS NUMBER(2);
BEGIN
    UPDATE INSTRUCTOR
        SET SALARY = SALARY + 5000
            WHERE SALARY < 65000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No instructor satisfied the condition');
    ELSIF SQL%FOUND THEN
        TOTAL_ROWS := SQL%ROWCOUNT;
        DBMS_OUTPUT.PUT_LINE(TOTAL_ROWS || ' instructors updated');
    END IF;
END;
/
```

Output:

```
3 instructors updated
```

5.1.1 Single-Row Implicit Cursors

The SELECT INTO statement is present in all implicit cursors that query data. It works only when a single row is returned by a select statement. You can select a column or list of columns in the SELECT clause and assign the row columns to individual variables or collectively to a record datatype. For example,

```
DECLARE
    -- For one-to-one assignment
    I_ID INSTRUCTOR.ID%TYPE;
    I_NAME INSTRUCTOR.NAME%TYPE;
    I_SALARY INSTRUCTOR.SALARY%TYPE;
    -- For group assignment
    TYPE INSTRUCTOR_RECORD IS RECORD
    (
        I_ID INSTRUCTOR.ID%TYPE,
        I_NAME INSTRUCTOR.NAME%TYPE,
        I_SALARY INSTRUCTOR.SALARY%TYPE
    );
    INS INSTRUCTOR_RECORD;
BEGIN
    -- Individual selection
    SELECT ID, NAME, SALARY
        INTO I_ID, I_NAME, I_SALARY
            FROM INSTRUCTOR
                WHERE ROWNUM < 2;
    DBMS_OUTPUT.PUT_LINE('Name: ' || I_NAME);
    -- Group selection
    SELECT ID, NAME, SALARY
        INTO INS
```

```
            FROM INSTRUCTOR
                WHERE ROWNUM < 2;
    DBMS_OUTPUT.PUT_LINE('Name: ' || INS.I_NAME);
END;
/
```

Output:

```
Name: Srinivasan
Name: Srinivasan
```

5.1.2 Multiple-Row Implicit Curosrs

There are two ways you can create multiple-row implicit cursors. The first is done by writing any DML statement in a PL/SQL block. DML statements are considered multiple-row implicit cursors, even though you can limit them to a single row. We already saw that in the UPDATE example. The second is to write an embedded query in a cursor FOR loop rather than defined in a declaration block. For example, the following program can be used to see tha names and the monthly salaries of different instructors:

```
BEGIN
    FOR ROW IN (SELECT NAME, SALARY FROM INSTRUCTOR) LOOP
        DBMS_OUTPUT.PUT_LINE('Name: ' || ROW.NAME);
  DBMS_OUTPUT.PUT_LINE('Monthly Salary: ' || TRUNC(ROW.SALARY/12));
    END LOOP;
END;
/
```

Output:

```
Name: Srinivasan
Monthly Salary: 5416
Name: Wu
Monthly Salary: 7500
Name: Mozart
Monthly Salary: 3333
Name: Einstein
Monthly Salary: 7916
Name: El Said
Monthly Salary: 5000
Name: Gold
Monthly Salary: 7250
Name: Katz
Monthly Salary: 6250
Name: Califieri
Monthly Salary: 5166
Name: Singh
Monthly Salary: 6666
Name: Crick
Monthly Salary: 6000
Name: Brandt
Monthly Salary: 7666
Name: Kim
Monthly Salary: 6666
```

5.2 Explicit Cursor

Explicit cursors are programmer-defined cursors for gaining more control over the context area. It should be defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns one or more rows. The prototype for declaring explicit cursor are as follows:

```
CURSOR cursor_name IS select_statement;
```

Four steps are required for using explicit curosrs:

1. Declaring the cursor for initializing the memory
2. Opening the cursor for allocating memory
3. Fetching the cursor for retrieving data
4. Closing the cursor to release allocated memory.

Static SELECT **Cursor**

Static SELECT statements return the same query each time with potentially different results. The result change as the data changes in the table or views.

For example, the following program can be used to iterate through department budgets:

```
DECLARE
    D_NAME DEPARTMENT.DEPT_NAME%TYPE;
    D_BUDGET DEPARTMENT.BUDGET%TYPE;
    CURSOR C_DEPT
    IS
        SELECT DEPT_NAME, BUDGET
            FROM DEPARTMENT;
BEGIN
    OPEN C_DEPT;
    LOOP
        FETCH C_DEPT INTO D_NAME, D_BUDGET;
        EXIT WHEN C_DEPT%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(D_NAME || ' ' || D_BUDGET);
    END LOOP;
    CLOSE C_DEPT;
END;
/
```

Output:

```
Biology 90000
Comp. Sci. 100000
Elec. Eng. 85000
Finance 120000
History 50000
Music 80000
Physics 70000
```

Dynamic SELECT **Cursor**

Dynamic SELECT statements act like parameterized subroutines. They run different queries each time, depending on the actual parameters provided when they are opened.

This can be achieved in two ways. An explicit cursor query can reference any variable in its scope. When you open an explicit cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

For example, the following program can be used to find the list of courses for a particular credit:

```
DECLARE
    C_TITLE COURSE.TITLE%TYPE;
    C_CREDIT COURSE.CREDITS%TYPE := 4;
    CURSOR C_COURSE
    IS
        SELECT TITLE
            FROM COURSE
                WHERE CREDITS = C_CREDIT;
BEGIN
    OPEN C_COURSE;
    LOOP
        FETCH C_COURSE INTO C_TITLE;
        EXIT WHEN C_COURSE%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(C_TITLE);
    END LOOP;
    CLOSE C_COURSE;
END;
/
```

Output:

```
Intro. to Biology
Genetics
Intro. to Computer Science
Game Design
Physical Principles
```

The value of C_CREDIT is evaluated and used during opening of the cursor. Even if we change the value after that, it will not affect the rows that are returned.

Relying on local variables can be confusing and more difficult to support the code. Another way can be defining cursors that accept formal parameters. You can create an explicit cursor that has formal parameters, and then pass different parameters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere that you can use a constant. Outside the cursor query, you cannot reference the formal cursor parameters. We can rewrite the previous program:

```
DECLARE
    C_TITLE COURSE.TITLE%TYPE;
    CURSOR C_COURSE
    (C_CREDIT COURSE.CREDITS%TYPE)
    IS
        SELECT TITLE
            FROM COURSE
                WHERE CREDITS = C_CREDIT;
BEGIN
    OPEN C_COURSE(4);
    LOOP
        FETCH C_COURSE INTO C_TITLE;
        EXIT WHEN C_COURSE%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(C_TITLE);
    END LOOP;
    CLOSE C_COURSE;
END;
/
```

Output:

```
Intro. to Biology
Genetics
Intro. to Computer Science
Game Design
Physical Principles
```

The benefit of this approach is that we can open the cursor multiple times using separate parameters to get different information. For example, the following program can identify the students in a certain department who are slacking off by taking lesser number of courses:

```
-- Create procedure for printing student information.
CREATE OR REPLACE
PROCEDURE PRINT_SLACKING_STUDENT
(S_DEPT IN STUDENT.DEPT_NAME%TYPE, S_AVG IN STUDENT.TOT_CRED%TYPE)
IS
    S_NAME STUDENT.NAME%TYPE;
    CURSOR C_STUDENT
    (DEPT STUDENT.DEPT_NAME%TYPE,
    AVERAGE STUDENT.TOT_CRED%TYPE)
    IS
        SELECT NAME
            FROM STUDENT
                WHERE DEPT_NAME = DEPT AND TOT_CRED < AVERAGE;
BEGIN
    OPEN C_STUDENT(S_DEPT, S_AVG);
    DBMS_OUTPUT.PUT_LINE('----------------');
    DBMS_OUTPUT.PUT_LINE('Slacking students from ' || S_DEPT);
    DBMS_OUTPUT.PUT_LINE('----------------');
    LOOP
        FETCH C_STUDENT INTO S_NAME;
        EXIT WHEN C_STUDENT%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(S_NAME);
    END LOOP;
    CLOSE C_STUDENT;
END;
/

-- Call procedure to find students
DECLARE
    S_AVG STUDENT.DEPT_NAME%TYPE;
BEGIN
    SELECT AVG(TOT_CRED) INTO S_AVG
        FROM STUDENT
            WHERE DEPT_NAME = 'Comp. Sci.';
    PRINT_SLACKING_STUDENT('Comp. Sci.', S_AVG);

    SELECT AVG(TOT_CRED) INTO S_AVG
        FROM STUDENT
            WHERE DEPT_NAME = 'Physics';
    PRINT_SLACKING_STUDENT('Physics', S_AVG);
END;
/
```

Output:

```
----------------
Slacking students from Comp. Sci.
----------------
Shankar
Williams
Brown
----------------
Slacking students from Physics
----------------
Snow
```

5.3 Cursor using FOR Loop

The cursor FOR loop is an elegant and natural extension of the numeric FOR loop in PL/SQL. The body of the for loop is executed for each row returned by the query. The benefit of using this loop is that Oracle handles the OPEN, FETCH, and CLOSE internally.

For example, the following program can be used to determine the yearly cost for each department in terms of salary of the instructors:

```
-- Declare function
CREATE OR REPLACE FUNCTION
TOTAL_SALARY(D_NAME INSTRUCTOR.DEPT_NAME%TYPE)
RETURN NUMBER
IS
    CURSOR C_INSTRUCTOR
    IS
        SELECT SALARY
            FROM INSTRUCTOR
                WHERE DEPT_NAME = D_NAME;
    TOTAL NUMBER := 0;
BEGIN
    FOR INS_ROW IN C_INSTRUCTOR
    LOOP
        TOTAL := TOTAL + INS_ROW.SALARY;
    END LOOP;
RETURN TOTAL;
END;
/

-- Call it from an anonymous block
BEGIN
    DBMS_OUTPUT.PUT_LINE(TOTAL_SALARY('Comp. Sci.'));
END;
/
```

Output:

```
232000
```

CHAPTER 6 TRIGGERS

Database triggers are specialized stored programs. As such, they are defined by very similar DDL rules. Likewise, triggers can call SQL statements and PL/SQL functions and procedures. You can choose to implement triggers in PL/SQL or Java.

Triggers differ from stored functions and procedures because you cannot call them directly. Database triggers are fired when a triggering event occurs in the database. This makes them very powerful tools in your efforts to manage the database. You are able to limit or redirect actions through triggers. Using triggers, you can:

- Control the behavior of DDL statements, as by altering, creating, or renaming objects
- Control the behavior of DML statements, like inserts, updates, and deletes
- Enforce referntial integrity, complex business rules, and security policies
- Audit information of system access and behavior by creating transparent logs

On the other hand, you cannot control the sequence of or synchronize calls to triggers, and this can present problems if you rely too heavily on triggers. The only control you have is to designate them as before or after certain events. Oracle 11g delivers compound triggers to help you manage larger events, like those triggering events that you would sequence.

There are risks with triggers. The risks are complex because while SQL statements fire triggers, triggers call SQL statements. A trigger can call a SQL statement that in turn fires another trigger. The subsequent trigger could repeat the behavior and fire another trigger. This creates cascading triggers. Oracle 11g and earlier releases limit the number of cascading trigger to 32, after which an exception is thrown. The trigger body can be no longer than 32,760 bytes. That is because trigger bodies are stored in `LONG` datatype columns.

There are five types of triggers: DDL triggers, DML triggers, Compound triggers, Instead-of triggers, and System or database event triggers. In this course, we will be focusing on DDL and DML triggers.

6.1 DDL Triggers

DDL triggers fire when you create, change, or remove objects in a database schema. They are useful to control or monitor DDL statements. An *instead-of create* table trigger provides you with a tool to ensure table creation meets your development standards, like including storage or partitioning clauses. You can also use them to monitor poor programming practices, such as when programs *create* and *drop* temporary tables rather than using Oracle collections. Temporary tables can fragment disk space and degrade database performance over time.

A list of data definition events that work with DDL triggers are show below:

Table 6.1: Available Data Definition Events

DDL Event	Description
ALTER	Changes object constraints, names, storage clauses, or structures.
ANALYZE	Computes statistics for the cost optimizer.
ASSOCIATE STATISTICS	Links a statistics type to a column, function, package, type, domain, index, or index type.
AUDIT	Enables auditing on an object or system.
COMMENT	Document column or table purposes.
CREATE	Creates objects in the database, like objects, privileges, roles, tables, users, and views.
DDL	Represents any of the primary data definition events. It effectively says any DDL event acting on anything.
DISASSOCIATE STATISTICS	Unlinks a statistic type from a column, function, package, type, domain index, or index type.
DROP	Drops objects in the database, like objects, privileges, roles, tables, users, and views.
GRANT	Grants privileges or roles to users in the database. The privileges enable a user to act on objects, like objects, privileges, roles, tables, users, and views.
NOAUDIT	Disables auditing on an object or system.
RENAME	Renames objects in the database, like columns, constraints, objects, privileges, roles, synonyms, tables, users, and views.
REVOKE	Revokes privileges or roles from users in the database. The privileges enable a user to act on objects, like objects, privileges, roles, tables, users, and views.
TRUNCATE	Truncates tables, which drops all rows from a table and resets the high-water mark to the original storage clause initial extent value. Unlike the DML DELETE statement, it cannot be reversed by a ROLLBACK command. You can use the new flashback to undo the change.

6.1.1 Event Attribute Functions

Few of the system-defined event attribute functions are:

- ORA_CLIENT_IP_ADDRESS
  Return: the client IP address as a VARCHAR2 datatype.

```
DECLARE
  IP_ADDRESS VARCHAR2(11);
BEGIN
  IF ORA_SYSEVENT = 'LOGON' THEN
    IP_ADDRESS := ORA_CLIENT_IP_ADDRESS;
  END IF;
END;
```

- ORA_DATABASE_NAME
  Return: the database name as a VARCHAR2 datatype.

```
DECLARE
```

```
  DATABASE VARCHAR2(50);
BEGIN
  DATABASE := ORA_DATABASE_NAME;
END;
```

- ORA_DES_ENCRYPTED_PASSWORD
  Return: the DES-encrypted password as a VARCHAR2 datatype.
  Equivalent to the value in the SYS.USER$ table PASSWORD column in Oracle 11g. Previously DBA_USERS or ALL_USERS view contained this information.

```
DECLARE
  PASSWORD VARCHAR2(60);
BEGIN
  IF ORA_DICT_OBJ_TYPE = 'USER' THEN
    PASSWORD := ORA_DES_ENCRYPTED_PASSWORD;
  END IF;
END;
```

- ORA_DICT_OBJ_NAME
  Return: an object name of the target of the DDL statement as a VARCHAR2 datatype. It also updates the

```
DECLARE
  DATABASE VARCHAR2(50);
BEGIN
  DATABASE := ORA_DICT_OBJ_NAME;
END;
```

- ORA_DICT_OBJ_NAME_LIST
  Parameter: a table of VARCHAR2(64) datatype containig list of object names touched by the triggering event.
  Return: the number of elements in the list as a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  NAME_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'ASSOCIATE_STATISTICS' THEN
    COUNTER := ORA_DICT_OBJ_NAME_LIST(NAME_LIST);
  END IF;
END;
```

- ORA_DICT_OBJ_OWNER
  Return: an owner of the object acted upon by the event as a VARCHAR2 datatype.

```
DECLARE
  OWNER VARCHAR2(30);
BEGIN
  OWNER := ORA_DICT_OBJ_OWNER;
END;
```

- ORA_DICT_OBJ_OWNER_LIST
  Parameter: a table of VARCHAR2(64) datatype containing the list of object owners where their statistics were analyzed by a triggering event.
  Return: the number of elements in the list indexed by a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  OWNER_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'ASSOCIATE_STATISTICS' THEN
    COUNTER := ORA_DICT_OBJ_OWNER_LIST(OWNER_LIST);
  END IF;
END;
```

- `ORA_DICT_OBJ_TYPE`
  Return: the datatype of the dictionary object changed by the event as a `VARCHAR2` datatype.

```
DECLARE
  OBJ_TYPE VARCHAR2(19);
BEGIN
  OBJ_TYPE := ORA_DICT_OBJ_TYPE;
END;
```

- `ORA_GRANTEE`
  Parameter: a table of `VARCHAR2(64)` datatypes containing the list of users granted privileges or roles by the triggering event.
  Return: the number of elements in the list indexed by a `PLS_INTEGER`. It also updates the list since it is passed by reference.

```
DECLARE
  USER_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'GRANT' THEN
    COUNTER := ORA_GRANTEE(USER_LIST);
  END IF;
END;
```

- `ORA_INSTANCE_NUM`
  Return: the current database instance number as a `NUMBER` datatype.

```
DECLARE
  INSTANCE NUMBER;
BEGIN
  INSTANCE := ORA_INSTANCE_NUM;
END;
```

- `ORA_IS_ALTER_COLUMN`
  Parameter: A column name.
  Return: True or False as a `BOOLEAN` datatype indicating whether the column has been altered or not.

```
DECLARE
  TYPE COLUMN_LIST IS TABLE OF VARCHAR2(32);
  COLUMNS COLUMN_LIST := COLUMN_LIST('CREATED_BY', 'LAST_UPDATED_BY')
   ;
BEGIN
  IF ORA_SYSEVENT = 'ALTER' AND ORA_DICT_OBJ_TYPE = 'TABLE' THEN
    FOR i IN 1 .. COLUMNS.COUNT LOOP
      IF ORA_IS_ALTER_COLUMN(COLUMNS(i)) THEN
        INSERT INTO LOGGING_TABLE
        VALUES(ORA_DICT_OBJ_NAME || '.' || COLUMNS(i) || ' changed.')
   ;
      END IF;
    END LOOP;
  END IF;
END;
```

- `ORA_IS_CREATING_NESTED_TABLE`
  Return: True or False value as a `BOOELAN` datatype indicating whether a nested table is created.

```
BEGIN
  IF ORA_SYSEVENT = 'CREATE' AND
     ORA_DICT_OBJ_TYPE = 'TABLE' AND
     ORA_IS_CREATING_NESTED_TABLE THEN
      INSERT INTO LOGGING_TABLE
      VALUES(ORA_DICT_OBJ_NAME || '.' || ' created with nested table.
   ');
  END IF;
END;
```

- `ORA_IS_DROP_COLUMN`
  Parameter: Column name.
  Return: True or False value as a `BOOLEAN` datatype indicating whether the column has been dropped or not.

```
DECLARE
  TYPE COLUMN_LIST IS TABLE OF VARCHAR2(32);
  COLUMNS COLUMN_LIST := COLUMN_LIST('CREATED_BY', 'LAST_UPDATED_BY')
   ;
BEGIN
  IF ORA_SYSEVENT = 'DROP' AND ORA_DICT_OBJ_TYPE = 'TABLE' THEN
    FOR i IN 1 .. COLUMNS.COUNT LOOP
      IF ORA_IS_DROP_COLUMN(COLUMNS(i)) THEN
        INSERT INTO LOGGING_TABLE
        VALUES(ORA_DICT_OBJ_NAME || '.' || COLUMNS(i) || ' changed.')
   ;
      END IF;
    END LOOP;
  END IF;
END;
```

- `ORA_IS_SERVERERROR`
  Parameter: An error number.
  Return: True or False value as a `BOOLEAN` datatype indicating whether the error is on the error stack or not.

```
BEGIN
  IF ORA_IS_SERVERERROR(4082) THEN
    INSERT INTO LOGGING_TABLE
    VALUES('ORA-0408 error thrown.');
  END IF;
END;
```

- `ORA_LOGIN_USER`
  Return: The current schema name as a `VARCHAR2` datatype.

```
BEGIN
  INSERT INTO LOGGING_TABLE
  VALUES(ORA_LOGIN_USER || ' is the current user.');
END;
```

- `ORA_PARTITION_POS`
  Return: The numeric position with the SQL text where you can insert a partition clause. Only available in an `INSTEAD OF CREATE` trigger.

```
DECLARE
  SQL_TEXT ORA_NAME_LIST_T;
  SQL_STMT VARCHAR2(32767);
  PARTITION VARCHAR2(32767) := 'partitioning_clause';
BEGIN
  FOR i IN 1 .. ORA_SQL_TXT(SQT_TEXT) LOOP
    SQL_STMT := SQL_STMT || SQL_TEXT(i);
  END LOOP;
  SQL_STMT := SUBSTR(SQT_TEXT, 1, ORA_PARTITIONING_POS - 1) || ' '
            || PARTITION || ' ' || SUBSTR(SQL_TEXT,
   ORA_PARTITION_POS);
  -- Add logic to prepend schema because this runs under SYSTEM.
  SQL_STMT := REPLACE(UPPER(SQL_STMT), 'CREATE TABLE ', 'CREATE TABLE
    ' || ORA_LOGIN_USER || '.');
  EXECUTE IMMEDIATE SQL_STMT;
END;
```

- ORA_PRIVILEGE_LIST
  Parameter: a table of VARCHAR2(64) datatypes containing the list of privileges or roles granted by the triggering event.
  Return: the number of elements in the list indexed by a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  PRIV_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'GRANT' OR
     ORA SYSEVENT = 'REMOVE' THEN
    COUNTER := ORA_PRIVILEGE_LIST(PRIV_LIST);
  END IF;
END;
```

- ORA_REVOKEE
  Parameter: a table of VARCHAR2(64) datatypes containing the list of users that had privileges or roles revoked by the triggering event.
  Return: the number of elements in the list indexed by a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  REVOKEE_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'REVOKE' THEN
    COUNTER := ORA_PRIVILEGE_LIST(REVOKEE_LIST);
  END IF;
END;
```

- ORA_SERVER_ERROR
  Parameter: the position on the error stack, where 1 is the top of the error stack.
  Return: an error number as a NUMBER datatype.

```
DECLARE
  ERROR NUMBER;
BEGIN
  FOR i IN 1 .. ORA_SERVER_ERROR_DEPTH LOOP
    ERROR := ORA_SERVER_ERROR(i);
  END LOOP;
END;
```

- ORA_SERVER_ERROR_DEPTH
  Return: the number of errors on the error stack as a PLS_INTEGER datatype.

- ORA_SERVER_ERROR_MSG
  Parameter: Position on the error stack, where 1 is the top of the error stack.
  Return: an error message text as a VARCHAR2 datatype.

```
DECLARE
  ERROR VARCHAR2(64);
BEGIN
  FOR i IN 1 .. ORA_SERVER_ERROR_DEPTH LOOP
    ERROR := ORA_SERVER_ERROR_MSG(i);
  END LOOP;
END;
```

- ORA_SERVER_ERROR_NUM_PARAMS
  Return: the count of any subsituted strings from error messages as a PLS_INTEGER datatype.

- ORA_SERVER_ERROR_PARAM
  Parameter: the position in an error message, where 1 is the first occurrence of a string in the error message.
  Return: an error message text as a VARCHAR2 datatype.

```
DECLARE
  PARAM VARCHAR2(32);
BEGIN
  FOR i IN 1 .. ORA_SERVER_ERROR_DEPTH LOOP
    FOR j IN 1 .. ORA_SERVER_ERROR_NUM_PARAMS(i) LOOP
      PARAM := ORA_SERVER_ERROR_PARAM(j);
    END LOOP;
  END LOOP;
END;
```

- ORA_SQL_TXT
  Parameter: A table of VARCHAR2(64) datatypes containing the substrings of the procedded SQL statement that triggered the event.
  Return: the number of element in the list indexed by a PLS_INTEGER datatype.

- ORA_SYSEVENT
  Return: the system event that was responsible for firing the trigger as a VARCHAR2 datatype.

```
BEGIN
  INSERT INTO LOGGING_TABLE
  VALUES(ORA_SYSEVENT || ' fired the trigger.');
END;
```

- ORA_WITH_GRANT_OPTION
  Return: True or False value as a BOOLEAN datatype indicating whether a privilege is granted with grant option.

```
DECLARE
  USER_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'GRANT' THEN
    COUNTER := ORA_GRANTEE(USER_LIST);
    IF ORA_WITH_GRANT_OPTION THEN
      FOR i IN USER_LIST.FIRST .. USER_LIST.LAST LOOP
        INSERT INTO LOGGING_TABLE
        VALUES(i || ' is granted privilege with grant option');
      END LOOP;
    END IF;
  END IF;
END;
```

- SPACE_ERROR_INFO
  Parameter: error number, error type, object owner, table space name, object name, sub object name.
  Return: True or False indicating whether the triggering event is related to an out-of-space condition or not. When the value is true, it also updates the variables since they are passed by reference.

```
DECLARE
  ERROR_NUMBER NUMBER;
  ERROR_TYPE VARCHAR2(12);
  OBJECT_OWNER VARCHAR2(30);
  TABLESPACE_NAME VARCHAR2(30);
  OBJECT_NAME VARCHAR2(128);
  SUBOBJECT_NAME VARCHAR2(30);
BEGIN
  IF SPACE_ERROR_INFO(ERROR_NUMBER, ERROR_TYPE,
                      OBJECT_OWNER, TABLESPACE_NAME,
                      OBJECT_NAME, SUBOBJECT_NAME) THEN
    INSERT INTO LOGGING_TABLE
    VALUES( /* implementation dependent */);
  END IF;
END;
```

6.1.2 Building DDL Triggers

The prototype for building DDL triggers is:

```
CREATE [OR REPLACE]
TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} ddl_event ON {DATABASE | SCHEMA}
[WHEN (logical_expression)]
[DECLARE]
  declaration_statements;
BEGIN
  execution_statements;
END [trigger_name];
```

You can use the INSTEAD OF clause only when auditing a creation event. Before triggers make sure the contents of the trigger body occur before the triggering DDL command, while after triggers run last.

Note that, table and trigger can share the same name. This is possible because there are two namespaces in Oracle databases: one for triggers and another for everything else.

Let's see an example of triggers. If we want to log CREATE, ALTER, and DELETE for a particular user (schema):

```
CREATE OR REPLACE
TRIGGER DDL_INFO_TRIGGER
BEFORE CREATE OR ALTER OR DROP ON SCHEMA
BEGIN
  DBMS_OUTPUT.PUT_LINE('Who did it? ' || ORA_DICT_OBJ_OWNER);
  DBMS_OUTPUT.PUT_LINE('What was the operation? ' || ORA_SYSEVENT);
  DBMS_OUTPUT.PUT_LINE('On what? ' || ORA_DICT_OBJ_NAME);
  DBMS_OUTPUT.PUT_LINE('On what type of object it was? ' ||
   ORA_DICT_OBJ_TYPE);
```

```
END;
/
```

6.2 DML Triggers

DML triggers can fire before or after INSERT, UPDATE, and DELETE statements. They can be statement-level or row-level activities.

- Statement-level triggers are executed once for each DML statement no matter how many rows are affected by the statement.
- Row-level triggers are executed for each row changed by a DML statement.

The prototype for building DML trigger is:

```
CREATE [OR REPLACE]
TRIGGER trigger_name
{BEFORE | AFTER}
{INSERT | UPDATE | UPDATE OF column1, [column 2 [, column(n+1)]] | DELETE
   }
ON table_name
[FOR EACH ROW]
[WHEN (logical_expression)]
[DECLARE]
  [PRAGMA AUTONOMOUS_TRANSACTION;]
  declaration_statements;
BEGIN
  execution_statements;
END [trigger_name];
```

The BEFORE or AFTER clause determines whether the trigger fires before or after the change is written to your local copy of the data. You can define a BEFORE or AFTER clause on tables but not views. While the prototype shows either an insert, update, update of (a column), or delete, you can also use an inclusion OR operator between the events. Using one OR between two events creates a single trigger that runs for two events. You can create a trigger that supports all four possible events with multiple inclusion operations.

You can use FOR EACH ROW clause to indicate that the trigger should fire for each row as opposed to once per statement. The WHEN clause acts as a filter specifying when the trigger fires.

6.2.1 Statement-Level Triggers

Statement-level triggers are also known as table-level triggers because they are triggered by a change to a table. They capture and process information when a user inserts, updates, or deletes one or more rows in a table. You can also restrict (filter) UPDATE statement triggers by constraining them to fire only when a specific column value changes. You can restrict the trigger by using a UPDATE OF clause. The clause can apply to a column name or a comma-delimited list of column names. You cannot use a WHEN clause in a statement-level trigger.

Consider that we want to keep a log of what DML statements are performed in our database. We have a table EVALUATIONS_LOG(EVENT_DATE, ACTION) to keep track of the actions happening on the EVALUATIONS table:

```
CREATE OR REPLACE
TRIGGER EVAL_CHANGE_TRACKER
AFTER INSERT OR UPDATE OR DELETE
ON EVALUATIONS
DECLARE
  CURR_ACTION EVALUATIONS_LOG.ACTION%TYPE;
BEGIN
  IF INSERTING THEN
```

```
    CURR_ACTION := 'Insert';
  ELSIF UPDATING THEN
    CURR_ACTION := 'Update';
  ELSIF DELETING THEN
    CURR_ACTION := 'Delete';
  END IF;
  INSERT INTO EVALUATIONS_LOG
  VALUES(SYSDATE, CURR_ACTION);
END;
/
```

6.2.2 Row-Level Triggers

Row-level triggers let you capture new and prior values from each row. This information can let you audit changes, analyze behavior, and recover prior data without performing a database recovery operation.

There are two pseudo-records when you use the `FOR EACH ROW` clause in a row-level trigger. They both refer to the columns referenced in the DML statement. The pseudo-records are composite variables; `NEW` and `OLD` are the pseudo-record variable names in the `WHEN` clause, and `:NEW` and `:OLD` are the bind variables in the trigger body. They differ because the trigger declaration and body are separate PL/SQL blocks. These pseudo-records can be used to access the columns that are changed by the DML statement:

- For an `INSERT` trigger, `OLD` contains no value, and `NEW` contains the new values.
- For an `UPDATE` trigger, `OLD` contains the old values, and `NEW` contains the new values.
- For a `DELETE` trigger, `OLD` contains the old values, and `NEW` contains no value.

Assume that when a student enrolls in a course, the credit for that course is added to their total credit. Let's also assume that, one student cannot take more than 180 credits. To make sure that the student does not violate this rule, we can use a trigger:

```
CREATE OR REPLACE
TRIGGER TOTAL_CREDIT_LIMIT_CHECKER
BEFORE INSERT ON TAKES
FOR EACH ROW
DECLARE
  COURSE_CREDIT COURSE.CREDITS%TYPE;
  CURRENT_TOT_CRED STUDENT.TOT_CRED%TYPE;
  NEW_TOT_CRED STUDENT.TOT_CRED%TYPE;
BEGIN
  SELECT TOT_CRED INTO CURRENT_TOT_CRED
  FROM STUDENT
  WHERE STUDENT.ID = :NEW.ID;
  SELECT CREDITS INTO COURSE_CREDIT
  FROM COURSE
  WHERE COURSE.COURSE_ID = :NEW.COURSE_ID;
  NEW_TOT_CRED := CURRENT_TOT_CRED + COURSE_CREDIT;
  IF NEW_TOT_CRED > 180 THEN
    RAISE_APPLICATION_ERROR(-2000, 'Registering for ' || :NEW:COURSE_ID
   || ' exceeds the total credit limit of 180.');
  ELSE
    UPDATE STUDENT
    SET TOT_CRED = NEW_TOT_CRED
    WHERE STUDENT.ID = :NEW.ID;
  END IF;
END;
/
```

Here, RAISE_APPLICATION_ERROR procedure is used to issue an user-defined error. It not only shows the error message, but also halts the execution of the INSERT statement.

Let's consider another example. Assume that you have developed a website where users can register by providing their username and password. The username can contain alphanumeric characters and underscore. If, by mistake, some user enters the username with whitespace in it, you want to replace them with underscore. The following program can be helpful:

```
CREATE OR REPLACE
TRIGGER SANITIZE_USERNAME
BEFORE INSERT OR UPDATE OF USERNAME ON USER
FOR EACH ROW
WHEN (REGEXP_LIKE(NEW.USERNAME, ' '))
BEGIN
  :NEW.USERNAME := REGEXP_REPLACE(:NEW.USERNAME, ' ', '_', 1, 1);
END;
/
```

The WHEN clause checks whether the value of the pseudo-field for the USERNAME column in the USER table contains a whitespace or not. If the condition is met, the trigger passes control to the trigger body. The trigger body has one statement; the REGEXP_REPLACE function takes a copy of the pseudo-field as an actual parameter. It changes any whitespace in the string to an underscore, and returns the modified value as a result. The result is assigned to the pseudo-field, and becomes the value in the INSERT/UPDATE statement. The reason we consider UPDATE statement here is because if it were only for the INSERT statement, the user could register and then update their profile to put whitespace in their usernames. This is an example of using a DML trigger to enforce a business policy of entering all usernames with alphanumeric characters and underscore.

Side Note: Sequences

Sequences are counting structures that maintain a persistent awareness of their current value. They are simple to create when you want them to start at 1 and increment by 1. The basic sequence also sets no cache, minimum, or maximum values and accepts both NOCYCLE and NOORDER properties. A sequence caches values by groups of 20 by default, but you can overwrite the cache size when creating the sequence or by altering it after creation. Following is the syntax for building a generic SEQUENCE:

```
CREATE SEQUENCE sequence_name
MINVALUE value
MAXVALUE value
START WITH value
INCREMENT BY value
CACHE value;
```

One example use case for sequences can be the following:

```
CREATE SEQUENCE STUDENT_SEQ
MINVALUE 1001
MAXVALUE 9999
START WITH 1
INCREMENT BY 1
CACHE 20;

CREATE OR REPLACE
TRIGGER STUDENT_ID_GENERATOR
BEFORE INSERT ON STUDENT
FOR EACH ROW
DECLARE
  NEW_ID STUDENT.ID%TYPE;
BEGIN
  SELECT STUDENT_SEQ.NEXTVAL INTO NEW_ID
    FROM DUAL;
  :NEW.ID := NEW_ID;
END;
/
```

This process is simplified in Oracle 11g:

```
CREATE OR REPLACE
TRIGGER STUDENT_ID_GENERATOR
BEFORE INSERT ON STUDENT
FOR EACH ROW
BEGIN
  :NEW.ID := STUDENT_SEQ.NEXTVAL;
END;
/
```

Many designs simply build these generic sequences and enable rows to be inserted by the web application interface. Some tables require specialized setup rows. These rows are inserted by administrators. They often use special primary key values below the numbering sequence assigned the regular application. When you have a table requiring manual setup rows, some developers leave the first 100 values and start the sequence at 101. Other applications leave more space and start the sequence at 1001. Both approaches are designed to provide your application with the flexibility to add new setup rows after initial implementation. This lets you isolate setup row values in a range different than ordinary applications data.

Sequences are typically built to support primary key columns in tables. Primary key columns impose a combination contraint on their values - they use both `UNIQUE` and `NOT NULL` constraints. During normal Ornline Transaction Processing (OLTP), some insertions are rolled back because other transactional components fail. When transactions are rolled back, the captured sequence value is typically lost. This means that you may see numeric gaps in the primary key column sequence values. Typically, you ignore small gaps. Larger gaps in sequence values occur during after-hours batch processing where you are performing bulk inserts into tables. Failures in batch processing typically involve operation staff intervention in conjunction with programming teams to fix the failure and process the data. Part of fixing this type of failure is resetting the next sequence value. While it would be nice to simply use an `ALTER` statement to reset the next sequence value, you cannot reset the `START WITH` number using an `ALTER` statement. YOu can reset every other criterion of a sequence, but you must drop and recreate the sequence to change the `START WITH` value.

There are three steps in the process to successfuly modify a sequence `START WITH` value: querying the primary key that uses the sequence to find the highest current value, dropping the existing sequence with the `DROP SEQUENCE sequence_name;` command, and recreating the sequence with a `START WITH` value one greater than the highest value in the primary key column. Naturally, the gap does not hurt anything, and you can skip this step, but as a rule, it is recommended to eliminate gaps during maintenance operations.

6.3 Order of Firing Triggers

Oracle allows more than one trigger to be created for the same timing point, but it has never guaranteed the execution order of those triggers. Oracle 11g trigger syntax now includes the `FOLLOWS` clause to guarantee execution order for triggers defined with the same timing point. The following example creates a table with two triggers for the same timing point:

```
CREATE TABLE TRIGGER_FOLLOWS_TEST
(
  ID NUMBER,
  DESCRIPTION VARCHAR2(50);
);

CREATE OR REPLACE
TRIGGER TRIGGER_FOLLOWS_TEST_TRG_1
```

```
BEFORE INSERT ON TRIGGER_FOLLOWS TEST
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('TRIGGER_FOLLOWS_TEST_TRG_1 - executed');
END;
/

CREATE OR REPLACE
TRIGGER TRIGGER_FOLLOWS_TEST_TRG_2
BEFORE INSERT ON TRIGGER_FOLLOWS TEST
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('TRIGGER_FOLLOWS_TEST_TRG_2 - executed');
END;
/
```

If we insert data into the table, there is no guarantee of the execution order. Oracle 11g provides `FOLLOWS` clause that can be used to specify the one trigger will follow another.

```
CREATE OR REPLACE
TRIGGER TRIGGER_FOLLOWS_TEST_TRG_2
BEFORE INSERT ON TRIGGER_FOLLOWS_TEST
FOR EACH ROW
FOLLOWS TRIGGER_FOLLOWS_TEST_TRG_1
BEGIN
  DBMS_OUTPUT.PUT_LINE('TRIGGER_FOLLOWS_TEST_TRG_2 - executed');
END;
/
```