A

CaoSys

Training Guide

PL/SQL for Beginners

Introduction

Objective

This guide is designed to give you a basic understanding of Oracle's procedural extension to SQL, PL/SQL. After reading the guide and working through the associated workbook you should be equipped with enough knowledge to enable you to produce and support PL/SQL programs of simple to medium complexity. Most major features of the language are covered in varying detail, from flow control statements and exception handling to cursors.

Who is this Guide for?

The guide is suitable for any Oracle developer using PL/SQL, this is anyone who works with an Oracle database.

Note that this guide is meant only as an introduction to PL/SQL and therefore much of the newer, more advanced features available in Oracle databases 8i, 9i and 10g are not covered.

Prerequisites

This guide assumes you have some knowledge of Relational Databases and Structured Query Language (SQL).

Programming knowledge from other languages is not essential but if you do have some programming experience, this will be of benefit.

For more information on SQL and SQL*Plus, download the *SQL & SQL*Plus for Beginners* guide from the CaoSys website.

Materials

This guide consists of 3 training booklets: -

1. This training guide
2. A booklet containing practical exercises and self-test questions
3. A discussion document covering some additional features

Contents

1 – Introduction to PL/SQL **9**
What is PL/SQL? 10
Client-Server Architecture 13
PL/SQL Features 14
Summary 16

2 – PL/SQL Fundamentals **18**
Basic PL/SQL Syntax 20
The PL/SQL Block 21
 Multiple Blocks 22
 Nested Blocks 23
 Block Types 24
Scope & Visibility 26
Comments 27
Variables 30
 Scalar Types 31
 Declaration 32
 Initialisation 33
 Type Anchoring 34
 Constants 36
Assignments & Expressions 37
PL/SQL From SQL*Plus 42
Viewing the Results 45
Flow Control 50
 The `IF` Statement 51
 Simple Loop 54
 The `FOR` Loop 56
 The `WHILE` Loop 58
 Nested Loops 59
 Labelling Loops 60
 Labelling Blocks 62
 The `GOTO` Statement 63
 PL/SQL Coding Style 64
 Summary 71
3 – SQL within PL/SQL **73**

DML not DDL 75
The SELECT Statement 76
The INTO Clause 77
The Implicit Cursor 78
UPDATE, INSERT and DELETE 79
The WHERE Clause 80
Transaction Control 82
Testing Outcome of a SQL Statement 84
Summary 86

4 - Exceptions 88
What is an Exception? 90
Types of Exception 91
Handling Exceptions 93
Exception Propagation 97
Creating Your Own Exceptions 99
Exceptions for an Oracle Error Code 100
Exceptions for your own Conditions 101
Summary 102

5 - Explicit Cursors 104
What is an Explicit Cursor? 106
How to use an Explicit Cursor 107
Declaring 108
Opening 109
Fetching 110
Closing 114
Cursor Attributes 115
A Complete Example 116
WHERE CURRENT OF 117
Cursor FOR Loop 119
Summary 122

6 – Stored Procedures & Functions 124
Stored Subprograms - An Overview 126
Procedures & Functions - The Difference 128

Creating and using a Stored Procedure 129
Creating and using a Stored Function 139
Handling Exceptions 143
Local Subprograms 147
A Few Guidelines 149
Benefits of Stored Subprograms 150
Summary 151

7 – Packages 153
What is a Package? 155
Packages and Scope 158
Creating and using a Package 159
Overloading Subprograms 166
Invoking Packaged Functions from SQL Statements 169
Privileges & Security 173
Oracle Provided Packages 176
Summary 177

8 – Triggers 179
What is a Trigger? 181
Trigger Types 183
Creating a Trigger 185
Restrictions on Triggers 190
Summary 191

Section One

Introduction to PL/SQL

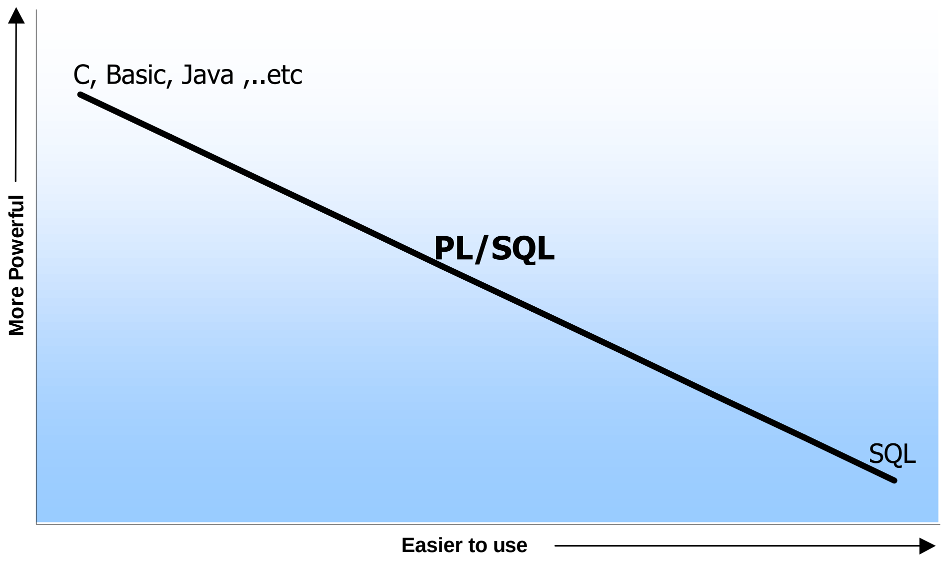
What is PL/SQL?

PL/SQL stands for Procedural Language SQL and it is an extension to SQL. SQL is what is known as a fourth generation language (kind of), in that you tell it what you want but do not tell it how to do it. This makes for very fast and easy software development. SQL on its own has its drawbacks though, in that it is not very flexible and it cannot be made to react differently to differing situations very easily, this is where PL/SQL comes in. PL/SQL adds procedural extensions to SQL traditionally found in third-generation languages such as C, Basic or Java. PL/SQL allows you to work with the database with the ease of SQL while giving you the power and flexibility of procedural constructs such as:-

Variables
Flow Control
Error Handling…etc.

What is PL/SQL?

The following chart shows that as ease of use increases with a programming language then power and flexibility reduce. C being at one end of the scale with SQL at the other and PL/SQL somewhere in between.



What is PL/SQL?

PL/SQL is the central programming language found in most of Oracle's Developments Tools, just as SQL is the core language used for database manipulation.

PL/SQL comes in two forms:-

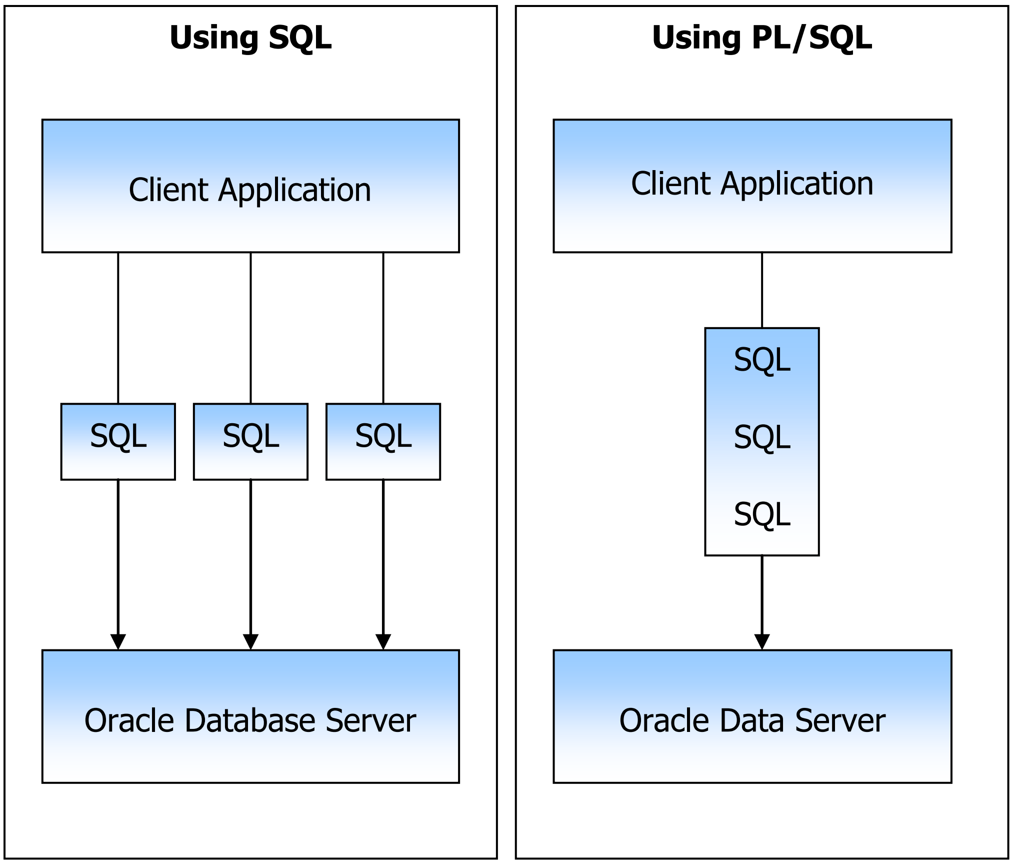
- Server Based - This is version 2 or above of PL/SQL. Typically PL/SQL programs are held as stored procedures/functions and packages within the database server itself. Execution of code is done on the server.
- Client Based - This is version 1.x of PL/SQL and is found in client-based tools such as Forms 4.5 and Reports 2.5. From Oracle Developer 6i+, the version of PL/SQL is the same on the client as it is on the server.

This guide concentrates on Server Based PL/SQL, although most of what is covered can be applied to Client Based PL/SQL.

Client-Server Architecture

Typically, a client-based application makes a request for information from the database using SQL. This will involve many network trips, thus affecting performance.

Using PL/SQL however, you can create a block of code containing several SQL statements and send them as a single request to the database. This improves performance.



PL/SQL Features

PL/SQL has many features, some of the major features are listed below:-

- **Variables & Constants** - Objects within PL/SQL that are used to store and manipulate values. Several scalar Datatypes are available, the more common ones being, `NUMBER`, `VARCHAR2`, `DATE` and `BOOLEAN`.
- **SQL** - All DML type SQL statements can be used directly within PL/SQL.
- **Flow Control** - PL/SQL supports flow control statements such as `IF`, `FOR`, `WHILE`. These allow for conditional actions, branching and iterative control.
- **Built-in Functions** - Most functions that are available in SQL can be used within a PL/SQL statement.
- **Cursor Management** - Cursors provide the ability to process data returned from multiple-row queries.

PL/SQL Features

PL/SQL features continued …

- **Block Structure** - PL/SQL programs are made up of blocks of code. Blocks are used to separate code into logical chunks. A PL/SQL program can be made up of several blocks and a block can in turn be made up of several blocks.
- **Exception Handling** - PL/SQL supports an elegant method for handling exceptions (errors) within code. The developer can also define his or her own exceptions.
- **Composite Types** - PL/SQL allows you to create composite datatypes, which typically relate to a row on a database table. You can also create array like structures called PL/SQL tables, VARARRY'S and Collection's.
- **Stored Code** - PL/SQL code can reside within the database server itself in the form of Packages, Procedures, Functions and Triggers. This makes the code available to all applications.

Summary

- PL/SQL is a procedural programming language designed specifically for working with Oracle Databases. It provides the ease of use of SQL whilst also giving the power and flexibility of a 3GL.
- PL/SQL supports Client-Server architecture and comes in two basic forms, Server based and Client based. PL/SQL is central to most of Oracle's development tools.
- PL/SQL has many features, most of which are covered in this course.

What is next?

Now we know what PL/SQL is, we can begin to take a look at the language and how to develop software with it.

Section Two

PL/SQL Fundamentals

PL/SQL Fundamentals

In this section we cover most of the fundamental aspects of the language. Specifically we will cover:-

- Basic PL/SQL Syntax
- The PL/SQL Block
- Commenting Code
- Variables
- Assignments and Expressions
- Scope and Visibility
- Viewing the Result
- PL/SQL Control Structures
- PL/SQL Coding Style

At the end of this section there will be several self-test questions to answer and exercises to perform

Basic PL/SQL Syntax

When entering any PL/SQL code, you should be aware of the following:-

- Free format language
- Statements can be split across many lines. Keywords cannot be split.
- Whitespace or an appropriate delimiter separates lexical units.
- Identifiers must start with an alpha-character, can be upto 30 characters in length and cannot be a reserved word (unless enclosed in double quotes)
- Character and Date literal are enclosed in single quotes.
- Each statement must be terminated with a semi-colon (;)

The PL/SQL Block

All PL/SQL programs are made up of Blocks. A PL/SQL program MUST contain at least one block. A Block is made up of PL/SQL statements enclosed within the keywords, BEGIN and END, for example:-

```
BEGIN
      INSERT INTO table (col) VALUES ('XX');
END;
```

If you wish to make any variable declarations you can also include a DECLARE section, for example:-

```
DECLARE
      v_number NUMBER;

BEGIN
      v_number := 10;
END;
```

An Exception Handler can also be included (covered later) using the EXCEPTION keyword, as follows:-

```
DECLARE
      v_number NUMBER;

BEGIN
      v_number := 10;

EXCEPTION
      WHEN OTHERS THEN
            DBMS_OUTPUT.put_line('Error');
END;
```

The PL/SQL Block

Multiple Blocks

A PL/SQL program can consist of several in-line blocks, for example:-

```
DECLARE
      v_number NUMBER;
```

Block 1

```
BEGIN
      v_number := 10;

EXCEPTION
      WHEN OTHERS THEN
            DBMS_OUTPUT.put_line('Error 1');
END;
```

Block 2

```
BEGIN
      v_number := 20;

EXCEPTION
      WHEN OTHERS THEN
            DBMS_OUTPUT.put_line('Error 2');
END;
```

Block 3

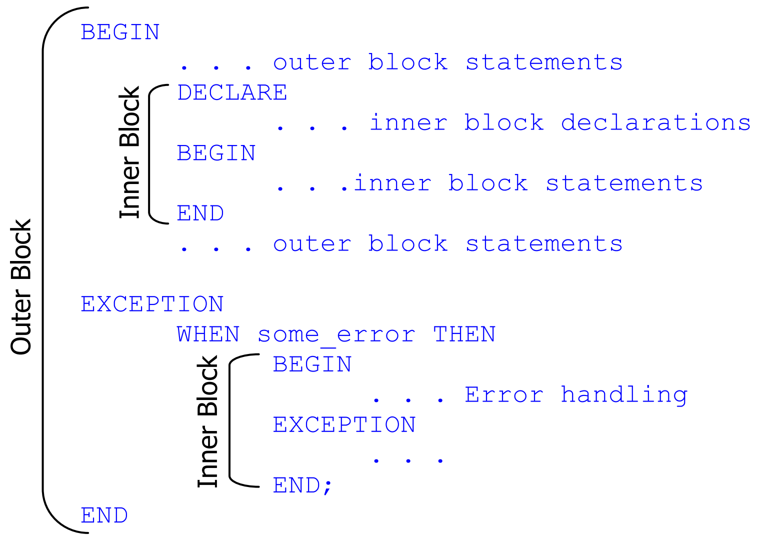
```
BEGIN
      v_number := 20;

EXCEPTION
      WHEN OTHERS THEN
            DBMS_OUTPUT.put_line('Error 3');
END;
```

Don not worry about what the above is doing, the code is simply used to demonstrate multiple blocks in a single piece of code.

The PL/SQL Block

Nested Blocks

Blocks can contain nested blocks; basically anywhere in a PL/SQL program where an executable statement can appear (i.e. between `BEGIN` and `END`) then a block can also appear. For example,



```
BEGIN
      . . . outer block statements
      DECLARE
            . . . inner block declarations
      BEGIN
            . . .inner block statements
      END
      . . . outer block statements

EXCEPTION
      WHEN some_error THEN
            BEGIN
                  . . . Error handling
            EXCEPTION
                  . . .
            END;
END
```

Blocks will typically be nested to break down a larger block into more logical pieces. Also, as we will discover later in the Exception Handling section, nested blocks allow for greater control of errors.

The PL/SQL Block

Block Types

There are several types of blocks:-

- **Anonymous Block** - Seen in the examples so far. These blocks have no name and are generally stored in a host file or entered directly into SQL*Plus and executed just once.
- **Named Blocks** - Very much the same as an Anonymous Block except the block is given a label.
- **Subprograms** - Packages, Procedures and Functions are blocks of code stored within the database. These blocks are executed via a specific call via the block name.
- **Triggers** - These are also named blocks that are stored within the database. Triggers are executed implicitly whenever the triggering event occurs.

The PL/SQL Block

Block Types - Examples

Named Blocks - Name is enclosed in << and >>:-

```
<<my_block>>
DECLARE
      v_number := 10;
BEGIN
      v_number := v_number * 10;
END;
```

Subprograms - Function to square a number:-

```
CREATE OR REPLACE FUNCTION square(p_number IN NUMBER)
IS
BEGIN
      RETURN p_number * 2;
END;
```

Triggers - These are also named blocks that fire in response to a database event.

```
CREATE OR REPLACE TRIGGER audit
      BEFORE DELETE ON items
      FOR EACH ROW
BEGIN
      INSERT INTO audit_table(item,description)
            VALUES(:old.item,:old.description);
END;
```

Scope and Visibility

The block in which an object is declared, including any nested blocks is the **scope** on an object.

```
DECLARE
      x NUMBER;

BEGIN
      DECLARE
            y NUMBER;

      BEGIN
            . . .
      END;

      . . .
END;
```

Inner Block
Scope of y

Outer Block
Scope of x

If the variable in the inner block was also called x as in the outer block, then whilst in the inner block, x would refer to the x declared in the current block. x in the outer block still exists but it has no **visibility**.

```
DECLARE
      x NUMBER;

BEGIN
      DECLARE
            x NUMBER;

      BEGIN
            . . .
      END;

      . . .
END;
```

x from outer block has no visibility

Comments

Comments can be added to your code in two ways:-

Single-line Comments

These comments start with two dashes, the comment continues until the end of the line, for example:-

```
-- Local declarations
v_number NUMBER ; -- A number variable
```

Multiple-line Comments

These comments begin with `/*` and end with `*/`, everything in between these delimiters is treated as a comment, for example:-

```
/* Determine what the item
   type is before continuing */
IF v_item_type = 'A' THEN
      . . .
```

Multi-line comments cannot be nested.

Comments

Comments should be used as much as possible. Good comments can make your code much easier to understand. Do not over-comment or include comments for comments sake. Try not to simply repeat what is already clear in your code, much of PL/SQL is quite readable and comments such as:-

```
-- Test if v_number is equal to 10
IF v_number = 10 THEN
      . . .
```

are useless and a waste of space. Try to explain the business logic behind your code. If you have produced a very complex program then your comments should include a good description of the algorithm involved.

I recommend you only ever single line comments, this allows you to use multi-line comments whilst debugging, you can easily comment out whole chunks of your program without worrying about illegal nesting of multi-line comments.

Comments

Different developers have different style for comments, it's really down to personal choice but here are a few pointers:-

- Always indent comments to the same level as your code
- Choose a low maintenance style, for example,

  High Maintenance

  ```
  -- ----------------- --
  -- This is a comment --
  -- ----------------- --
  ```

  Low Maintenance

  ```
  --
  -- This is a comment
  --
  ```

- Don't restate the obvious
- Try to comment each logical block of code
- Try to maintain a modification history within the code, either at the top or bottom of a program
- Remember, other developers may need to read your comments

Variables

Variables are an essential part of any programming language. A variable is simply a location in memory where you can store data of different types.

Variables can be read or set during execution of a PL/SQL block, values from the database can be stored in variables as can literals and values of other variables.

All variables must be declared before they can be used. The declaration of a variable specifies the type and size of the data that the variable will hold.

There are a number of different types of variable within PL/SQL, these types fall into one of the following categories:-

- Scalar Types - covered next
- Composite Types - covered later

Two other categories exist, Reference Type and LOB Types, these are not covered.

Variables - Scalar Types

A scalar type holds a single value of a specific type. The main datatypes supported by PL/SQL are those which correspond directly to the types held within the database, BOOLEAN types are also supported.

Datatype	Max Size	Default Size	Description
NUMBER (width,scale)	38	38	Numeric values rounded to a whole number unless a scale is given, i.e. NUMBER(5,2) means a numeric values 5 digits in length allowing for 2 decimal places
VARCHAR2 (width)	32767 *		Variable length character data
CHAR (width)	32767 **		Fixed length character data
DATE			Valid date values
BOOLEAN			Boolean values TRUE and FALSE

Several other subtypes are available, these are basically made up from the types above. Other types include, BINARY_INTEGER, DEC, DECIMAL, FLOAT, DOUBLE PRECISION, INT, INTEGER, NATURAL, NUMERIC, PLS_INTEGER and POSITIVE. We will not cover these other types.

* A VARCHAR2 database column is only 2000 bytes long in Oracle7 and 4000 bytes long in Oracle8+

** 255 for PL/SQL v1.x

Variables - Declaration

Before a variable can be used within your code, you must first declare it. All declarations must appear in the DECLARE section of your block. Variable declaration has the following syntax:-

```
varname TYPE [CONSTANT] [NOT NULL] [:= value];
```

where varname is the name of the variable, the CONSTANT keyword specifies this variable as a constant (covered later), the NOT NULL keyword ensures the value can never be NULL and the final part is a default value; some examples:-

```
DECLARE
      v_anumber1 NUMBER(10);
      v_anumber2 NUMBER := 100;
      v_astring VARCHAR2(1) NOT NULL := 'Y';
```

All the above are valid declarations. The following is invalid,

```
v_a_string VARCHAR2(1) NOT NULL;
```

The above declaration is not valid because the variable is constrained with NOT NULL and no default value has been given.

The DEFAULT keyword can also be used if preferred, i.e.

```
v_astring2 VARCHAR2(1) NOT NULL DEFAULT 'Y';
```

Variables - Initialisation

If you declare a variable without specifying a default/initial value, you may be wondering what value it does contain, PL/SQL automatically initialises all variables without a default value to NULL. The is unlike many other languages such as C where a variable may just contain garbage once declared.

Variables - Type Anchoring

Another way to declare a variables type is using the Type Anchoring method. If you know the type of a variable is directly related to a column on a table within the database then you can anchor the type of your variable to the type on the database, for example, suppose we have a table called `items`, this has a `NUMBER` column called `item_id`, then within your PL/SQL program you could:-

```
v_item_id     NUMBER;
```

This is inflexible, if the type of the database was ever to change, you would have to change your code, you could on the other hand anchor the type as follows:-

```
v_item_id items.item_id%TYPE;
```

Where `items` is the table name and `item_id` is the column name. This will always ensure that `v_item_id` is the correct type.

Variables - Type Anchoring

You can also anchor a variables type to another variable in the same program, for example:-

```
v_item_id    items.item_id%TYPE;
v_new_item   v_item_id%TYPE;
```

I recommend you use type anchoring wherever possible, it makes for more flexible code and can avoid bugs that are introduced through incompatible type use.

Variables - Constants

A variable can be declared as a constant value by using the `CONSTANT` keyword within the variable declaration.

A constant allows you to assign a value to an identifier that can then be referenced throughout the program. Any attempt to change a constant value will cause an error to occur.

I suggest using constants within your program rather than using magic numbers (hardcoded values). For example,

```
DECLARE
      c_bad_status CONSTANT NUMBER := 5;
BEGIN
      IF v_item_status = c_bad_status THEN
      . . .
```

is much better than,

```
IF v_item_status = 5 THEN
. . .
```

Assignments & Expressions

As already seen, a variable can be declared with a default value, this is actually an assignment. Assignments can appear anywhere within a block of code, not just the DECLARE section. An Assignment is the storing of a value in a variable. Assignment has the following basic syntax:-

```
identifier := expression;
```

Where identifier is the variable name and expression is the value you wish to store. The expression can be a literal, another variable or any other type of complex expression. Here are some basic examples:-

```
v_counter := 0;
v_name := 'This is the name';
v_date := TO_DATE('10-JAN-99','DD-MON-RR');
v_processed := TRUE;
```

The key part of an assignment is the assignment operator, `:=`, notice that this is different to the equality operator `=`.

Assignments & Expressions

The expression part of an assignment is what is know as an RVALUE, this means that an expression cannot be used as a statement on its own and it must appear on the right-hand side of an assignment.

An expression can be anything that will be evaluated to give a result, this could be some math, on literals and/or other variables, it could also be the result of calling a function, in fact, almost anything that can be done in SQL can be used as an expression in PL/SQL. Here are some examples of expressions:-

```
5 * 10
v_count + 50
first_digit(names_cur.full_name)
SYSDATE
```

When used in assignments, they are as follows:-

```
v_total := 5 * 10;
v_count := v_count + 50;
v_first_name := first_name(names_cur.full_name);
v_today := SYSDATE;
```

When evaluating numeric expressions, normal operator precedence applies.

Assignments & Expressions

Boolean expressions are always evaluated to TRUE or FALSE (and sometimes NULL). A Boolean expression is referred to as a condition when used in control flow statements (such as `IF`). All of the following are Boolean expressions:-

```
5 > 10
X < Y
A = B AND D > E
```

When used in assignments, they are as follows:-

```
v_boolean := 5 > 10;
v_x_is_bigger_than_y := X < Y;
x_continue := A = B AND D > E;
```

Assignments & Expressions

Remember the logical (relational) operators from the SQL course, here they are again:-

Logical Operators

Operator	Meaning
=	equal to (equality)
!=	not equal to (inequality)
<=	less than or equal
>=	more than or equal
<	less than
>	more than

Three other operators exist AND, OR and NOT, these operators accept Boolean values and return Boolean values based on the following truth table.

NOT	**TRUE**	**FALSE**	**NULL**
	FALSE	TRUE	NULL
AND			
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL
OR			
TRUE	TRUE	TRUE	NULL
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Any of the above operators can be used as part of an expression.

Assignments & Expressions

There are some more advanced operators available within PL/SQL, these are `LIKE`, `IS NULL`, `BETWEEN` and `IN`, these work exactly the same as they do in SQL (the `WHERE` clause), here are some examples:-

```
v_valid_type := v_item_type LIKE 'QB%';
v_not_set := IS NULL v_number;
v_in_range := v_id BETWEEN 10 AND 50;
v_in_list := v_id IN(10,30,50,70,90);
```

Almost all functions that can be used within SQL can also be used in an expression. Note that the `DECODE` function as well as any group functions CANNOT be used.

Expressions can be as complex as you like, although I recommend that you try to keep them understandable (and maintainable), if an expression is 10 lines long then you probably want to re-think your code.

PL/SQL From SQL*Plus

Before we move on, let us first see how we can execute a PL/SQL program. The easiest and most common method of running a PL/SQL program which is an anonymous block is via SQL*Plus.

There a two basic ways to execute PL/SQL from SQL*Plus:-

- Enter PL/SQL directly into the SQL buffer and run it.
- Create a PL/SQL script file and then execute that file from SQL*Plus.

We will take a quick look at both. We will be using SQL*Plus for executing PL/SQL throughout the rest of this course.

PL/SQL From SQL*Plus

The first method of executing PL/SQL from SQL*Plus is by entering PL/SQL commands directly into the SQL buffer, this is done much the same as it is for entering SQL commands, with one major difference, when entering a SQL statement, you terminate the statement with a blank line or semicolon, this does not work for PL/SQL programs, you must terminate the PL/SQL program with a dot '`.`', you can then execute the PL/SQL block by entering `RUN` or '`/`' at the prompt, for example:-

```
SQL> DECLARE
  2     v_number NUMBER := 0;
  3   BEGIN
  4     v_number := v_number + 10;
  5   END;
  6   .                  - This will terminate the block
SQL> /                   - This will execute the block
```

You can edit PL/SQL in the usual way with SQL*Plus commands or by typing `ed` to invoke the editor.

If the PL/SQL code is executed successfully you should see:-

```
PL/SQL procedure successfully completed
```

PL/SQL From SQL*Plus

The second method of executing PL/SQL from SQL*Plus is by creating PL/SQL script files. These files are entered and executed in the exact same way as SQL script files. Here is an example to refresh your memory:-

```
SQL> DECLARE
  2     v_number NUMBER := 0;
  3    BEGIN
  4     v_number := v_number + 10;
  5    END;
  6    .
SQL> SAVE myfile.pls
SQL> CLEAR BUFFER
SQL> GET myfile.pls
SQL> /
SQL> START myfile.pls
SQL> @myfile.pls
```

In the above example, a file called `myfile.pls` is created, the buffer is cleared and the file is retrieved, it is then executed three times.

Viewing Errors

In any errors are present in your program (compile time errors), you can view errors by querying the `USER_ERRORS` database view, a more convenient method is to use the SQL*Plus command:-

```
SQL> show err[ors]
```

Viewing the Results

Okay, so we've written our first PL/SQL program, it all worked okay, but we saw nothing, we got no messages back at all. How do we get output back from PL/SQL?

There are four main ways to get output from PL/SQL, these are:-

- Insert information into a table and query back the table
- Use a SQL*Plus bind variable
- Use the Oracle supplied package `DBMS_OUTPUT` to send output to the screen
- Use the Oracle supplied package `UTL_FILE` to send out to a host file. This is an advanced topic and will be covered at the end of the course.

Viewing the Results - Method 1

The first method of viewing the result of a PL/SQL program is by inserting information into a database table.

First, create a table to store the information, this can be called anything you like, i.e.

```
SQL> CREATE TABLE my_debug
 2   (       date_created       DATE
 3   ,       text               VARCHAR2(500));
```

Now for the PL/SQL:-

```
SQL> DECLARE
 1           l_x NUMBER := 0;
 2   BEGIN
 3           INSERT INTO my_debug
 4           VALUES (SYSDATE,'Before='||TO_CHAR(l_x));
 5           l_x := l_x + 10;
 6           INSERT INTO my_debug
 7           VALUES (SYSDATE,'After='||TO_CHAR(l_x));
 8   END;
 9   .
SQL> /
```

Now query back the table, ordering the data by the date column:-

```
SQL> SELECT        text
 1   FROM          my_debug
 2   ORDER BY      date_created;
```

You should now see the following:-

```
TEXT
--------------
Before=0
After=10
```

Viewing the Results - Method 2

Another way to view the result of your PL/SQL program is by using SQL*Plus bind variables.

First, you need to declare your SQL*Plus bind variable, this is done with the `VARIABLE` command, `VARIABLE` has the following syntax:-

```
VAR[IABLE] name [NUMBER|CHAR[(n)]|VARCHAR2[(n)]]
```

So, we declare a variable:-

```
SQL> VAR result NUMBER
```

Now the PL/SQL:-

```
SQL> DECLARE
 1          l_x NUMBER := 0;
 2    BEGIN
 3          l_x := l_x + 10;
 4          :result := l_x;
 5    END;
 6    .
SQL> /
```

Now, print the value in SQL*Plus:-

```
SQL> print result
```

You should now see:-

```
RESULT
--------------
10
```

Viewing the Results - Method 3

Yet another way to view the result of your PL/SQL program is by using an Oracle supplied package (packages are covered later), `DBMS_OUTPUT`. This package contains several procedures that can be used to print output on screen (as well as providing other facilities).

Before you can use DBMS_OUTPUT you must first enable output in SQL*Plus with the following command:-

```
SET serverout[put] ON|OFF [SIZE n]
```

Size specifies the size of the output buffer in bytes, if this buffer overflows, you will get the following error:-

```
ORA-20000: ORU-10027: buffer overflow, limit of n bytes
```

So, to switch on output and specify a size:-

```
SQL> SET serverout ON SIZE 1000000
```

I suggest always setting the size to a large number, this way you are less likely to encounter problems with overflows. It is a good idea to include the above line a file called `login.sql`, this is executed each time you enter SQL*Plus.

Viewing the Results - Method 3

Once output is enabled you can use the following procedures:-

Procedure	Description
`PUT`	Append text to the current line of the output buffer
`NEW_LINE`	Put and end-of-line marker into the output buffer, this effectively flushes the buffer.
`PUT_LINE`	Append text and an end-of-line marker to the current line in the output buffer. This also flushes the buffer.

When calling a packaged procedure from within PL/SQL, you must qualify the procedure name with the package name, so, to print a Hello World message:-

```
DBMS_OUTPUT.put_line('Hello World');
```

Is the same as:-

```
DBMS_OUTPUT.put('Hello');
DBMS_OUTPUT.put_line(' World');
```

And is also the same as:-

```
DBMS_OUTPUT.put('Hello World');
DBMS_OUTPUT.new_line;
```

NOTE
Output is only given once the PL/SQL code has finished.

Method 4 is covered later.

Flow Control

PL/SQL provides many structures that control the flow of a program. PL/SQL provides the following conditional expressions:-

- `IF` statement
- `EXIT` statement

The following looping structures are also provided:-

- Simple LOOP
- `FOR` loop (Numeric FOR Loop)
- `WHILE` loop
- Loop Labels
- Block Labels
- The `GOTO` statement
- CURSOR FOR loop (covered later)

Flow control structures are an essential part of any third generation programming language, it is these constructs combined with variables and subprograms that give PL/SQL its power over SQL. We will now take a look at each of the flow control statements mentioned above.

Flow Control

The `IF` Statement

The `IF` statement allows PL/SQL to take different **actions** based on certain **conditions**, its syntax is very similar to the `IF` statement found in many other programming languages.

Condition

We have covered conditions already but just to recap, a condition is the result of a Boolean expression, for example, the condition:-

```
X > Y
```

is TRUE if X is more than Y, otherwise FALSE.

Action

An action is one or more PL/SQL statements. An action can be anything from inserting into a table to performing more `IF` statements.

Flow Control

The IF Statement

The IF statement has the following syntax:-

```
IF condition1 THEN
      action1;
[ELIF condition2 THEN
      action2;]
[ELSE
      action3;]
END IF;
```

For example,

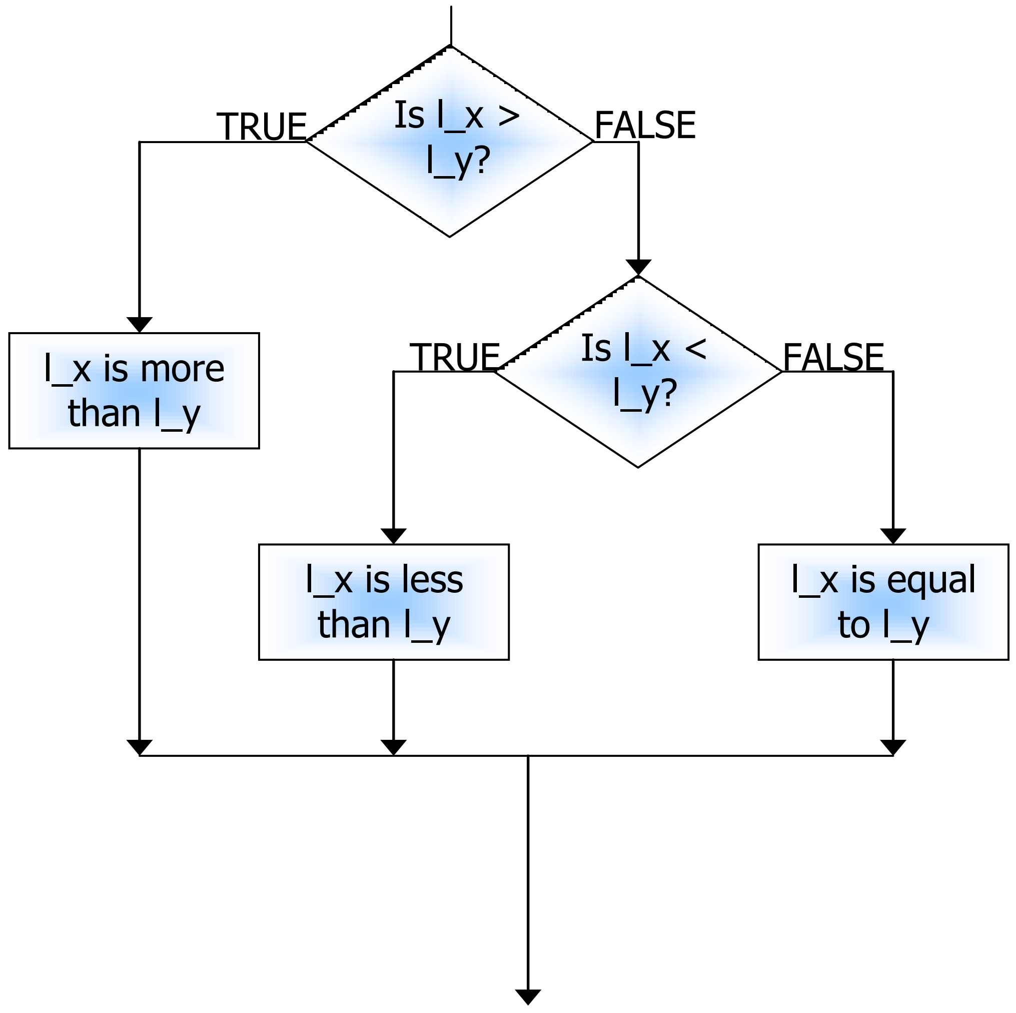
```
DECLARE
      l_x NUMBER := 10;
      l_y NUMBER := 20;
BEGIN
      IF l_x > l_y THEN
            DBMS_OUTPUT.put_line('X is greater than Y');
      ELIF l_x < l_y THEN
            DBMS_OUTPUT.put_line(X is less than Y');
      ELSE
            DBMS_OUTPUT.put_line('X is equal to Y')
      END IF;
END;
```

The above PL/SQL program declares two variables, `l_x` and `l_y`, they are defaulted to 10 and 20 respectively, then the program tests whether `l_x` is greater than, less than or equal to `l_y`. The `ELSE` part of the statement is like a catch all, it basically says, if `l_x` does not match anything then perform this action.

Flow Control

The `IF` Statement

Here is a flowchart of the previous example:-



Flow Control

Simple Loop

A loop is a section of code that needs to be executed a number of times. PL/SQL has many kinds of loops, the most basic form of loop, a simple loop has the following syntax:-

```
LOOP
      statements ..
END LOOP;
```

In the above loop, the *statements* are executed forever, this is because the loop has no way out, there is no condition applied to the loop that will stop it. To ensure the loop has a condition that can stop it you must use the `EXIT` statement. This has the following syntax:-

```
EXIT [WHEN condition];
```

where *condition* can be any kind of Boolean expression, for example:-

```
LOOP
      l_x := l_x + 10;

      EXIT WHEN l_x > 100;
END LOOP;
```

In the above example, the loop will continue adding 10 to `l_x` until it is more than 100.

Flow Control

Simple Loop

The EXIT statement can also be used inside an IF statement:-

```
LOOP
      l_x := l_x + 10;

      IF l_x > 100 THEN
            EXIT;
      END IF;
END LOOP;
```

Flow Control

The FOR Loop

A FOR (numeric FOR) loop is similar to the simple loop we have already seen, the main difference is the controlling condition is found at the start of the loop and the number of iterations is known in advance, whereas with a simple loop the number of iterations is not known and is determined by the loop condition (the EXIT statement). The FOR loop has the following syntax:-

```
FOR variable IN [REVERSE] start_value .. end_value
LOOP
        statements ..
END LOOP;
```

The variable is the controlling variable, its value starts as start_value and increments until it reaches end_value. The REVERSE keyword can be used to execute to loop in the opposite direction.

variable is only in scope so long as the loop is running, if you need to preserve its value then you must set a declared variable to its value within the loop.

Flow Control

The FOR Loop

Here are a few examples of using a numeric FOR loop.

To display numbers 1 to 10:-

```
FOR counter IN 1 .. 10
LOOP
      DBMS_OUTPUT.put_line(counter);
END LOOP;
```

To display numbers 10 to 1:-

```
FOR counter IN REVERSE 1 .. 10
LOOP
      DBMS_OUTPUT.put_line(counter);
END LOOP;
```

The start_value and end_value do not have to be numeric literals, they can be any expression that results in a numeric value, for example:-

```
DECLARE
      l_days      NUMBER;
BEGIN
      l_days := l_date2 - l_date1;
      FOR counter IN 0..l_days
      LOOP
            INSERT INTO date_table
                  (date) VALUES
                  (l_date2 + counter);
      END LOOP;
END;
```

The above example calculates the number of days between two dates and inserts a row into a table for each date.

Flow Control

The `WHILE` Loop

The `WHILE` loop again is very similar to the simple loop, even more so than the `FOR` loop. A `WHILE` loop will continue so long as a condition is true. Its syntax is as follows:-

```
WHILE condition
LOOP
      statements . .
END LOOP;
```

The *condition* can be any Boolean expression that can be used in an `IF` or `EXIT` statement, statements can be any sequence of PL/SQL statements. Using the previously seen simple loop example, it can be written using a `WHILE` loop:-

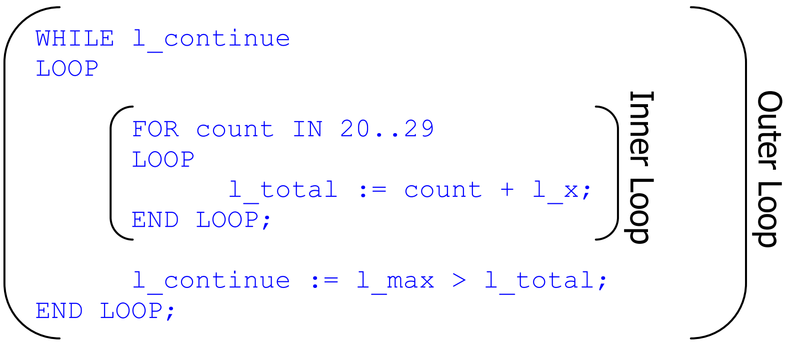
```
WHILE l_x <= 100
LOOP
      l_x := l_x + 10;
END LOOP;
```

The above example says, continue to increment `l_x` so long as `l_x` is less than or equal to 100. Notice the difference in condition compared to the `EXIT` statement, this condition says continue so long as condition is true, whereas the `EXIT` statement says stop processing if a condition is true.

Flow Control

Nested Loops

Loops can be nested to almost any number of levels. You can nest different kinds of loops within other types, for example, you may have a numeric `FOR` loop which is nested within a `WHILE` loop, for example:-



```
WHILE l_continue
LOOP

      FOR count IN 20..29
      LOOP
            l_total := count + l_x;
      END LOOP;

      l_continue := l_max > l_total;
END LOOP;
```

The above example has two nested loops, a `FOR` within a `WHILE`. The `WHILE` will continue so long as the variable `l_continue` is true and the `FOR` loop will iterate from 20 to 29 (10 times). After processing the `FOR` loop, `l_continue` is set to the result of the Boolean expression `l_max > l_total`, if this is `FALSE` then the `WHILE` will end.

Flow Control

Labelling Loops

Let's say you have written some code which makes use of nested loops, how do you make the program terminate an outer loop from an inner loop?

This can be done is two ways:-

- Raise an exception, we cover exceptions later
- Label the loop and use the `EXIT` statement, we cover this method next.

Flow Control

Labelling Loops

You can give loops a label, you do this by prefixing the loop with the following:-

```
<< label-name >>
```

for example,

```
<<my_loop>>
WHILE l_continue
LOOP
      . . . .
END LOOP my_loop;
```

If you have a nested loop, you can exit the outer loop from within the inner loop by using the `EXIT` statement with a loop label, for example:-

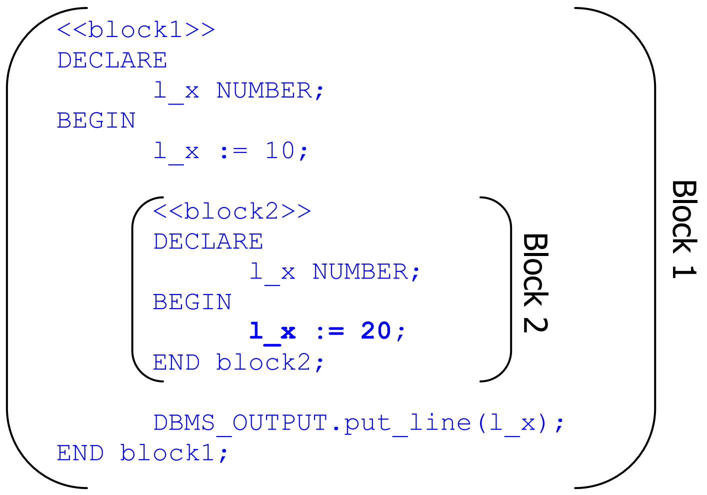
```
<<outer>>
WHILE l_continue
LOOP

      <<inner>>
      FOR count IN 20..29
      LOOP
            l_total := count + l_x;

            EXIT outer WHEN l_total > l_overflow;

      END LOOP inner;

      l_continue := l_max > l_total;
END LOOP outer;
```

The above example checks the value of `l_total` within the `FOR` loop, if it is greater than the variable `l_overflow` then the outer loop is exited. The label name in the `END LOOP` in optional.

Flow Control

Labelling Blocks

You can label a block in the same way you can label a loop, for example:-



```
<<block1>>
DECLARE
      l_x NUMBER;
BEGIN
      l_x := 10;

      <<block2>>
      DECLARE
            l_x NUMBER;
      BEGIN
            l_x := 20;
      END block2;

      DBMS_OUTPUT.put_line(l_x);
END block1;
```

The variable `l_x` in block2 refers to `l_x` declared in block 2, `l_x` declared in block1 has no visibility. You can make block 1's `l_x` visible by qualifying `l_x` with the block label:-

```
<<block1>>
DECLARE
      l_x NUMBER;
BEGIN
      l_x := 10;

      <<block2>>
      DECLARE
            l_x NUMBER;
      BEGIN
            block1.l_x := 20;
      END block2;

      DBMS_OUTPUT.put_line(l_x);

END block1;
```

Flow Control

The GOTO Statement

Using labels, you can use the GOTO statement. GOTO allows you to directly jump (branch) to a particular section of code. GOTO is unconditional and final, there is no return. Here is an example:-

```
DECLARE
      l_x NUMBER := 30;
      l_y NUMBER := 20;
BEGIN
      IF l_x >= l_y THEN
            GOTO skip_calc;
      END IF;

      l_x := l_y;
      DBMS_OUTPUT.put_line('l_x is now same as l_y');

      <<skip_calc>>
      DBMS_OUTPUT.put_line(l_x);

END;
```

The GOTO in the above block simply skips the calculation part of the code if l_x is greater than l_y. You cannot use the GOTO statement to pass control into the middle of a sub-block.

I recommend you avoid using GOTO, overuse can make programs very hard to follow and leads to unstructured code.

PL/SQL Coding Style

As far as the fundamentals of PL/SQL go, that is it, we have covered pretty much everything you need. Before we summarise the session and begin the first set of exercises, we will have a quick look at PL/SQL coding styles.

PL/SQL Coding Style

Good programming style is hard to define because everyone has their own personal tastes, but, there are guidelines you can follow to ensure that your coding is understandable by others and even yourself if you return to the code months later.

Programming style does not really affect how a program runs, it does affect how the program is debugged, enhanced and even coded in the first place. Coding style involves good use of variable names, whitespace and comments, consider the following block of code:-

```
declare
a1 number:=10;
a2 number:=20;
a3 number;
begin if a1 > a2 then a3:=a1-a2; elsif a1<a2 then
a3:=a2-a1; else a3:=0; end if; end;
```

The above code is VERY hard to read and understand, this is incredibly BAD coding style. Imagine if 6 months later you had to change this code,….forget it!!!!!

PL/SQL Coding Style

Okay, we've seen some bad style (or no style), so lets re-write the same code in a good style:-

```
DECLARE
      l_first      NUMBER := 10;
      l_second     NUMBER := 20;
      l_difference NUMBER;
BEGIN
      --
      -- Determine difference between two numbers
      --
      IF l_first > l_second THEN
            l_difference := l_first - l_second;
      ELSIF l_first < l_second THEN
            l_difference := l_second - l_first;
      ELSE
            l_difference := 0;
      END IF;
END;
```

Which would you prefer to come back too?

PL/SQL Coding Style

Coding Style Guidelines:-

Style of comments

Comment should be used everywhere where some kind of explanation is required, usually at the start of each logical block of code or procedure. It is also good practice to comment each of the variables declared within a block. As said earlier, do not over comment.

Variable Names

Choose short and meaningful names for your variables where you can. the variable `l_difference` is much better than `n3`. Also, prefix your variables with a single letter which describes the type of variable, I usually use the following prefixes:-

Prefix	Used on	Example
l_ and g_	Local and global variables	l_counter, g_status
v_	Another commonly used prefix for program variables	v_counter
c_	Constant declaration	c_cancel_flag
p_	Parameter	p_output_file
t_	User-defined type	t_employee
tb_	PL/SQL Table	tb_employees
r_	PL/SQL record	r_employee_info
_cur (suffix)	Cursror	employee_cur

PL/SQL Coding Style

Variable Names

You can of course use any kind of prefixes you like, but you should pick a style and stick too it. Some organisations have their own in-house guidelines on style, if so, try your best to stick to these.

Using a good prefix (or suffix) along with a meaningful name can make code more self-documenting and much easier to read and understand.

Capitalisation

PL/SQL is not a case sensitive language, so case has no affect on the program itself, thus you are not forced to use a particular case. This means you can use capitalisation to your advantage. Good use of case would be:-

- Use upper case for ALL reserved words (`BEGIN`, `END`, `DECLARE`, `IF`,…etc)

PL/SQL Coding Style

Capitalisation

- Use lowercase for all variables, column names and database objects. Some developers prefer to capitalise the first letter of each word in a variable, e.g. `l_CountUpdates`.
- Package member functions and procedures in lowercase with the package name in uppercase, e.g. `DPOPPOPR.process`
- SQL keywords in uppercase (`SELECT`, `DELETE`, `WHERE`,…etc)
- Built-in functions in uppercase (`NVL`, `SUBSTR`,…etc)

Whitespace

Whitespace is very important when it comes to readability. You should use whitespace as follows:-

- Always indent logical blocks, i.e. indent code within a `BEGIN` and `END`, always indent the contents of a loop, likewise an `IF` statement. Indents should really be around 4 spaces.

PL/SQL Coding Style

Whitespace

- Separate blocks of code or groups of statements with a blank line.
- Try to line up sections of multiple lines, i.e. The datatypes of variable within the `DECLARE` section.

Good programming style is something that comes with experience, the more you code, the better your style will be (it could of course get worse, if you yourself understand a language better, you may be less inclined to follow a good style because it looks okay to you, avoid this). But, you need to ensure you start as you mean to go on.

Summary

Phew!!! That was a lot of information to take in. To summarise then, we have learnt:-

- Basic syntax of PL/SQL
- The PL/SQL block
- Multiple Blocks
- Nested Blocks
- Block Types - Anonymous, Named, Subprograms and Triggers
- Scope and Visibility
- Comments - Single and Multiple line
- Variables - Scalar types, Declaration, Initialisation, Type Anchoring
- Constants
- Assignments and Expressions
- PL/SQL from SQL*Plus
- Viewing the Results
- Flow Control - Branching, Looping, Nested Loops, Labelling Loops, Labelling Blocks and `GOTO`
- PL/SQL coding Style

Now lets test our knowledge!!!

What is Next?

Section 3 deals with using SQL from with PL/SQL. We look at constructing a `SELECT` statement and using it within PL/SQL, we will also look at transaction control and explain what an Implicit Cursor is?

Section Three
SQL within PL/SQL

SQL within PL/SQL

In this section we cover how you use SQL statements within a PL/SQL program. Specifically we cover:-

- DML not DDL
- The `SELECT` Statement
- The `INTO` Clause
- The Implicit Cursor
- `UPDATE`, `INSERT` and `DELETE`
- The `WHERE` clause
- Transaction Control
- Testing Outcome of a SQL Command

At the end of this section there will be several self-test questions to answer and exercises to perform

DML not DDL

In you remember back to the SQL course, DML stands for *Data Manipulation Language* and allows you to query and change data within the database, whereas DDL stands for *Data Definition Language* and allows you to modify the structure of the database.

PL/SQL allows you to directly use any DML statement within a block of code, it does not allow the direct use of DDL commands, for example, you can construct a `SELECT` statement within PL/SQL but not a `CREATE TABLE` statement.

NOTE

Oracle provides a package called `DBMS_SQL` which allows you to create dynamic SQL and PL/SQL, using this package it is possible to perform DDL within PL/SQL. This is quite an advanced topic and is not covered on this course.

The `SELECT` Statement

Constructing a `SELECT` statement within PL/SQL is very simple, it is almost the same as creating a `SELECT` from within SQL*Plus with one major difference, take a look at the following `SELECT` statement:-

```
SELECT      ename
FROM        emp
WHERE       sal = (SELECT MAX(sal) FROM EMP);
```

The above code will select the highest paid employee and display their name. This is fine for SQL*Plus but it will not work in PL/SQL, why?, well the reason is that once the data has been `SELECT`ed then it has nowhere to go, remember we said that PL/SQL does not support direct screen output.

So, we have to tell PL/SQL where to put the data once selected, we do this with the `INTO` clause.

The INTO Clause

Taking the previous example, we would construct the statement using INTO as follows:-

```
SELECT      ename
INTO        <variable>
FROM        emp
WHERE       sal = (SELECT MAX(sal) FROM emp);
```

The above code will do the same as the previous example but this time the data returned will be stored in a variable. Variables can be any valid/suitable PL/SQL variable or SQL*Plus bind variable. So, as a complete example:-

```
DECLARE
      l_emp_name emp.ename%TYPE;
BEGIN
      SELECT      ename
      INTO        l_emp_name
      FROM        emp
      WHERE       sal = (SELECT MAX(sal) FROM emp);

      DBMS_OUTPUT.put_line(l_emp_name);
END;
```

The INTO is mandatory, a syntax error will occur if you omit it. A SELECT statement **MUST** return one and only one row, if no rows or more than one row is returned then an error will occur. We will discuss how we handle these errors later.

The Implicit Cursor

All data selected using SQL statements within PL/SQL is done using something called a cursor, there are two basic types of cursor, explicit (covered later) and implicit. An Implicit cursor is where you use a `SELECT` statement without the specific use of cursors (i.e. no cursor keywords), PL/SQL creates an implicit cursor for you.

An Implicit cursor works as follows:-

- Open the cursor
- Fetch from the cursor to get data
- Fetch again to check if any more rows are found

The term cursor is quite confusing at first, simply think of a cursor as a term given to the retrieval of data within PL/SQL, or in other words, a `SELECT` statement. Explicit cursors are used to `SELECT` more than one row and are covered later.

UPDATE, INSERT and DELETE

You can use the SQL statements UPDATE, INSERT and DELETE in PL/SQL in the exact same way as you would in SQL*Plus, for example:-

```
BEGIN
      -- Clear down debug table
      DELETE debug;

      -- Give salary increase
      UPDATE      emp
      SET   sal = sal * 1.15;

      -- Log salary increase is debug table
      INSERT INTO debug (text)
            VALUES ('Give everyone a 15% pay increase');

END;
```

The above example first of all clears down the debug table, then gives all employees a 15% pay increases, then a row is inserted into the debug table.

You can use INSERT, UPDATE and DELETE without restriction is PL/SQL.

The WHERE Clause

The WHERE clause in an integral part of any UPDATE, DELETE or SELECT statement. The WHERE clause determines what rows are affected/selected by the statement. A WHERE clause generally consists of one or more conditions that determine if a row is to be SELECTed, UPDATEd or DELETEd. Any statement that can have a WHERE clause generally takes the form:-

```
<SELECT|UPDATE|DELETE>
[WHERE condition];
```

Within the WHERE clause you can reference any appropriate PL/SQL variable, constant or SQL*Plus bind variable, for example:-

```
DECLARE
      l_find_job VARCHAR2(10) := 'PROGRAMMER';
BEGIN
      UPDATE emp
      SET job = 'DEVELOPER'
      WHERE job = l_find_job;
END;
```

The above example declares a variable called l_find_job, this is then referenced within the UPDATE statement to determine which employees need their job title changing.

The WHERE Clause

Be very careful when using PL/SQL variables within a WHERE clause, you must ensure that you do not use a variable with the same name as a column on the table, for instance, using the previous example, if we had written it as follows:-

```
DECLARE
      job VARCHAR2(10) := 'PROGRAMMER';
BEGIN
      UPDATE emp
      SET job = 'DEVELOPER'
      WHERE job = job;
END;
```

What do think would happen here? **ALL** rows on the emp table would have been updated because PL/SQL would determine that the `WHERE` clause is saying "where job on emp is equal to job on emp", this will be true for all rows on the table, therefore all rows will be updated.

Transaction Control

The COMMIT and ROLLBACK statements control transactions. A transaction is not determined by a block. For example:-

```
BEGIN
      UPDATE emp
      SET sal = sal * 1.15
      WHERE TRUNC(hiredate) <
            TRUNC(ADD_MONTHS(SYSDATE,-36));

      COMMIT;

      UPDATE emp
      SET sal = sal * 1.10
      WHERE TRUNC(hiredate) <
            TRUNC(ADD_MONTHS(SYSDATE,-24));
      COMMIT;

END;
```

The above code gives all employees who were hired more than 3 years ago a 15% pay increase, it then gives all employees hired more than 2 years ago a 10% pay increase. This is a single PL/SQL block with two transactions.

In some cases the host environment affects transactions created/started in a PL/SQL block, i.e. if you execute a PL/SQL block within PL/SQL without COMMITting the transaction, when you leave SQL*Plus the transaction is automatically committed.

Transaction Control

You can also make use of the `SAVEPOINT` statement within PL/SQL, for example:-

```
BEGIN
      DELETE debug;
      SAVEPOINT deleted_debug;

      DELETE transactions;
      ROLLBACK TO deleted_debug;

      COMMIT;
END;
```

You can also mix transactions within PL/SQL and the host environment, for example,

```
SQL>  BEGIN
 2          DELETE debug;
 3          SAVEPOINT deleted_debug;
 4          DELETE transactions;
 5    END;
 6    /
PL/SQL procedure successfully completed.

SQL> ROLLBACK TO deleted_debug
SQL> COMMIT;
```

In this example, the transaction is started within a PL/SQL block, but the host environment (in this case SQL*Plus) finished it off.

Testing Outcome of a SQL Statement

Whenever you issue a SQL statement in a PL/SQL block, whether it is a `SELECT` statement or some other DML statement, PL/SQL creates an implicit cursor, this cursor is automatically given an identifier of `SQL`. Attached to this identifier are a number of attributes that can be queried to determine the outcome of the SQL command. The following attributes are available:-

Attribute	**Description**
`SQL%ROWCOUNT`	The number of rows processed by the SQL statement
`SQL%FOUND`	TRUE if at least one row was processed by the SQL statement, otherwise FALSE
`SQL%NOTFOUND`	TRUE is no rows were processed by the SQL statement, otherwise FALSE.

Note that when a `SELECT` statement fails (i.e. no rows or more than one row) then an exception is raised, exceptions are covered in the next section. These attributes are commonly used to test the outcome on an `UPDATE`, `INSERT` or `DELETE`.

Testing Outcome of a SQL Command

If you issue DML statement and it affects no rows, then PL/SQL does not consider this a failure, so no exceptions are raised. Therefore you need a method of determining if your DML has actually done anything, this is where SQL attributes are useful. For example, suppose we have an `UPDATE` statement that should display some message if no rows are updated, here is how you do it:-

```
DECLARE
      l_rows_updated NUMBER;
BEGIN
      UPDATE some_table
      SET processed = 'Y'
      WHERE processed = 'N';

      IF SQL%NOTFOUND THEN
            DBMS_OUTPUT.put_line('No rows updated');
      ELSE
            l_rows_updated := SQL%ROWCOUNT;
            DBMS_OUTPUT.put_line('Updated :'||
                        TO_CHAR(l_rows_updated));
      END IF;
END;
```

Also, notice the use of `SQL%ROWCOUNT` to display the number of rows updated. This code could also have been written using the `SQL%FOUND` attribute simply by reversing the logic.

Summary

To summarise then, we have seen that you can use SQL within PL/SQL in almost the same way as you can in SQL*Plus. Note that you can't directly issue a DDL statement, only DML (including `SELECT`). All SQL commands use what is known as an implicit cursor.

- You can perform DML commands within PL/SQL in almost the same way as you can from SQL*Plus. DDL commands can't directly be used.
- The `SELECT` Statement requires an `INTO` clause and must return one and only one row.
- Implicit Cursors are created for all SQL commands within PL/SQL
- Transactions can be controlled using the `COMMIT` and `ROLLBACK` commands.
- You can test the outcome of a SQL command using SQL cursor attributes.

What is Next?

In section 4 we look at exceptions. We describe what an exception is and how they are handled. We also take a quick look at user-defined exceptions.

Section Four
Exceptions

Exceptions

This section deals with exceptions. We will be covering the following:-

- What is an Exception?
- Types of Exception
- Handling Exceptions
- Exception Propagation
- Creating your own Exceptions

At the end of this section there will be several self-test questions to answer and exercises to perform

What is an Exception?

An Exception is an identifier in PL/SQL that is used to trap for an abnormal condition. During the execution of a block, if an error occurs then an exception is raised, the block will terminate and control will pass to the exception handler if present.

Types of Exception

There are two types of exception:-

- Pre-defined - PL/SQL comes with several predefined exceptions, these are directly associated with Oracle error codes.
- User-Defined - These are declared by the programmer and can be associated with an Oracle error code that has no exception or they can be wholly your own error condition.

Types of Exception

PL/SQL comes with many pre-defined exceptions, these are directly associated with an Oracle error code. The exception provides a much easier way to trap for certain conditions, i.e. trapping for `NO_DATA_FOUND` is easier and more intuitive that checking for Oracle error code -1403. Here is a list of the more common exceptions:-

Exception	Oracle Error	Occurs when ...
DUP_VAL_ON_INDEX	-1	Attempt to store a duplicate key
NO_DATA_FOUND	-1403	`SELECT` statement returns no rows
VALUE_ERROR	-6502	Arithmetic, truncation, conversion error
INVALID_CURSOR	-1001	Invalid cursor operation such as closing an already closed cursor
TOO_MANY_ROWS	-1422	If a `SELECT` statement returns more than one row
INVALID_NUMBER	-1722	Attempt to convert a non-numeric value to a numeric

Handling Exceptions

Exceptions are handled through the use of Exception Handlers. An Exception handler is a section of PL/SQL code found at the end of a block labelled EXCEPTION. Within this section of code you use the WHEN statement to handle specific exceptions. For example:-

```
DECLARE
      l_string VARCHAR2(2);
BEGIN
      l_string := 'ABC';
END;
```

In the above example, we declare a variable l_string of type VARCHAR2 which is 2 in length, then we try to store a 3-digit string in it, this automatically raises the exception VALUE_ERROR which will give the following output:-

```
declare
 *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 4
```

Once the exception is raised the block terminates and the program ends, you have no control of what happens once this error occurs.

Handling Exceptions

A much better way to handle this problem is by using an Exception Handler, so, using the previous example but with an Exception Handler this time:-

```
DECLARE
      l_string VARCHAR2(2);
BEGIN
      l_string := 'ABC';
EXCEPTION
      WHEN value_error THEN
            DBMS_OUTPUT.put_line
                  ('Could not store value');
END;
```

this would give the following output:-

```
Could not store value
```

When the error occurs this time, control is passed to the `EXCEPTION` section where the `WHEN value_error` statement catches the error and displays a more meaningful message. You now have full control over what happens once an error occurs, you could for example, write some information to an external file or insert a row into another table, you can pretty much do anything you like.

Handling Exceptions

There is a special exception handler called `OTHERS`, this can be used with the `WHEN` statement to trap **ALL** exceptions not already catered for. For example:-

```
DECLARE
      l_name emp.ename%TYPE;
BEGIN
      SELECT ename
      INTO l_name
      FROM emp
      WHERE empno = l_empno;

EXCEPTION
      WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.put_line
                  ('Could not find employee');
      WHEN OTHERS THEN
            DBMS_OUTPUT.put_line
                  ('A Fatal Error Occurred');
END;
```

In this example, we a explicitly checking for `NO_DATA_FOUND`, but also, if **ANY** other error occurs, we want to simply display the message `'A Fatal Error Occurred'.`

Handling Exceptions

There are two very useful functions that can be used in conjunction with Exception Handlers:-

- SQLCODE - Returns the error number associated with the exception.
- SQLERRM - Returns the complete error message for the exception, including the error code.

Putting these to use in the previous example:-

```
DECLARE
      l_name emp.ename%TYPE;
BEGIN
      SELECT ename
      INTO l_name
      FROM emp
      WHERE empno = l_empno;

EXCEPTION
      WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.put_line
                  ('Could not find employee');
      WHEN OTHERS THEN
            IF SQLCODE = -1422 THEN
                  DBMS_OUTPUT.put_line
                        ('Too many rows returned');
            ELSE
                  DBMS_OUTPUT.put_line
                        ('Fatal Error : '||SQLERRM);
            END IF;
END;
```

As we can see in the above example, if the WHEN OTHERS code is executed then error code is checked and one of two messages is displayed.

Exception Propagation

Exception Propagation is the way exceptions are passed to the enclosing block if they are left un-handled. For example, if an error occurs in the following code within the inner block, then the exception is passed to the enclosing block where it is then handled:-

```
DECLARE
      l_string1 VARCHAR2(3);
BEGIN
      l_string1 := 'ABC';

      DECLARE
            l_string2 VARCHAR2(3);
      BEGIN
            l_string2 := 'ABCD';

      END;

EXCEPTION
      WHEN value_error THEN
            DBMS_OUTPUT.put_line('Error');

END;
```

Exception raised here and passed to enclosing block where it is handled

Be aware that if an exception is not handled by any handler then the exception will propagate out to the host environment, thus completely stopping execution of your PL/SQL program.

Exception Propagation

You may want certain actions to be performed if an error occurs but you may still want the program to continue, you do this by enclosing your code in its own block with its own exception handler, if the statement fails, you catch the exception and then continue with the rest of the program as normal, for example:-

```
DECLARE
      l_string VARCHAR2(3);

BEGIN
      l_string := 'ABC';                Exception raised here
      BEGIN
            l_string := 'ABCD';

      EXCEPTION
            WHEN VALUE_ERROR THEN          Handled here
                  DBMS_OUTPUT.put_line
                        ('Could not store value');
      END;
                              Execution continues here
      l_string := 'DE';
END;
```

In the above example, if an exception is raised when setting the value of `l_string` to `ABCD` then the exception is handled because this line of code is enclosed in its own block with its own handler, once handled, execution continues at the first line of code after the block.

Creating your own Exceptions

You can easily create your own exceptions. These user-defined exceptions can be directly related to an Oracle error code that currently has no exception or they can be your own error conditions.

Creating your own Exceptions

Exceptions for an Oracle Error Code

There a literally hundreds of errors that can occur, only a few have exceptions defined for them, you can create exceptions for these codes very easily. You do this in two steps, first declare your exception name, then associate your declared name with the Oracle Error code, for example, to create an exception called `e_fetch_failed` which should occur when Oracle Error -1002 occurs you would do the following:-

```
DECLARE
      e_fetch_failed EXCEPTION;
      PRAGMA EXCEPTION_INIT(e_fetch_failed,-1002);
BEGIN
      FETCH cursor INTO record;
WHEN
      WHEN fetch_failed THEN
            . . .
```

If this FETCH fails then error -1002 is raised automatically which is now associated with e_fetch_failed

This method is much easier to read and understand than the alternative which would be to have a `WHEN OTHERS` and then check `SQLCODE` for `-1002`.

Creating your own Exceptions

Exceptions for your own Conditions

You may have written a program that does a number of things, one of those things might be checking if a value has exceeded another value, if this condition is true then you may want to raise your own exception thus trapping for the condition. You do this by declaring your exceptions and then explicitly raising that exception with the RAISE command. For example:-

```
DECLARE
      e_too_many EXCEPTION;
BEGIN
      IF l_count > l_max_count THEN
            RAISE e_too_many;
      END IF;
      . . .
EXCEPTION
      WHEN e_too_many THEN
            . . .
```

In the above example, we first declare an exception called e_too_many, we then check if l_count is more than l_max, if it is then we RAISE e_too_many. You can create exceptions for practically anything.

Summary

We have seen that Exceptions are basically PL/SQL's method of handling errors that occur during execution of a block.

- There a two types of Exception, pre-defined and user-defined
- Exceptions are handled with exception handlers within your code.
- Exceptions always propagate to the enclosing block if left un-handled.
- You can create your own exceptions for Oracle Error codes and your own error conditions.

What is Next?

In section 5 we will cover explicit Cursors, we will describe what an explicit cursor is and how to use them.

Section Five

Explicit Cursors

Explicit Cursors

In this section we will learn about Explicit Cursors, we will cover:-

- What is an Explicit Cursor?
- How to use an Explicit Cursor . . .
  - Opening
  - Fetching From
  - Closing
  - Cursor Attributes
  - A Complete Example
  - `WHERE CURRENT OF` clause
- Cursor FOR Loops

What is an Explicit Cursor?

An Explicit Cursor is a named construct within PL/SQL that is used for retrieving rows from the database. There are two types of cursor:-

- Implicit - We have seen these already in section 3. These are generally a standalone `SELECT` (or other DML) statements within PL/SQL, they can only return a single row.
- Explicit - These are used to fetch more than one row from the database.

How to use an Explicit Cursor

Using Explicit Cursors is a little more complex than implicit cursors. Basically, there are 4 steps that need to taken:-

- Declare - A cursor must be declared before it can be used
- Open - A declared cursor needs to be opened before any data can be retrieved
- Fetch - Once open, rows can be fetched from the cursor
- Close - When you have finished with the cursor, you must ensure you close it.

These steps can be carried out manually one at a time or they can be almost automated by using a Cursor `FOR` Loop that we will cover later.

Explicit Cursor - Declaring

The declaration of a cursor, names it and defines the query that will be used. For example,

```
CURSOR employees_cur
IS
      SELECT      empno
      ,           ename
      FROM  emp
      WHERE job = 'CLERK';
```

The above code declares a cursor called `employees_cur`, the `SELECT` statements is parsed (validated) at this point but it is not executed. The cursor must de declared in the `DECLARE` section of your block.

You can also use parameters in your cursor as follows:-

```
CURSOR employees_cur(p_job VARCHAR2)
IS
      SELECT      empno
      ,           ename
      FROM  emp
      WHERE job = p_job;
```

The above code declares a cursor that accepts 1 parameter, `p_job`, this parameter is then referenced within the `SELECT` statement.

The `SELECT` statement can be any valid `SELECT` statement.

Explicit Cursor - Opening

Once declared, before you can retrieve any data you must first open the cursor, opening the cursor actually executes the query and places a pointer just before the first row, for example:-

```
OPEN employees_cur;
```

Any parameters passed are evaluated at this point. Here is the OPEN statement again but this time with a parameter:-

```
OPEN employees_cur('CLERK');
```

or

```
OPEN employees_cur(l_employee_job);
```

All rows are determined and are now called the *active set*. The *active set* is now ready for fetching.

You can use a cursor attribute called ISOPEN to test whether a cursor is already open, for example:-

```
IF employees_cur%ISOPEN THEN
      DBMS_OUTPUT.put_line('Already open');
ELSE
      OPEN employees_cur(l_employee_job);
END IF;
```

The above code checks to see if a cursor is open already before attempting the OPEN statement.

Explicit Cursor - Fetching

Once open, rows can be fetched from the *active set*, this is done with the `FETCH` statement. Data `FETCH`ed from a cursor **MUST** be fetched into a variable, one variable is required for each column returned.

```
DECLARE
      CURSOR employees_cur(p_job VARCHAR2)
      IS
            SELECT      empno
            ,           ename
            FROM   emp
            WHERE  job = p_job;

      l_empno emp.empno%TYPE;
      l_ename emp.ename%TYPE;

BEGIN

      OPEN employees_cur('CLERK');

      FETCH employees_cur
            INTO l_empno,l_ename;
      . . .
```

In the above example, the cursor is opened and a row is retrieved, the columns are stored in the variables `l_empno` and `l_ename`.

Explicit Cursor - Fetching

A more convenient way to store the columns returned in a cursor is to use a PL/SQL record. A PL/SQL record is a complex datatype that is similar to a structure in C. PL/SQL records can be created in two ways, manually which we will cover later and by row anchoring. Row anchoring is similar to type anchoring that we looked at earlier but instead of anchoring a type to a particular column on a table, you anchor the type to all columns on a table or all columns returned by a cursor. For example, using the previous example:-

```
DECLARE
      CURSOR employees_cur(p_job VARCHAR2)
      IS     SELECT      empno
             ,           ename
             FROM   emp
             WHERE  job = p_job;

      r_emp_rec employees_cur%ROWTYPE;
BEGIN
      OPEN employees_cur('CLERK');

      FETCH employees_cur INTO r_emp_rec;
      . . .
```

The important part of the above code is the declaration of the record, `r_emp_rec`, notice the use of `%ROWTYPE`

Explicit Cursor - Fetching

Once data has been `FETCH`ed into a record, it can be read back using dot notation, using the previous example, we would display both columns from the `FETCH` as follows:-

```
DBMS_OUTPUT.put_line(r_emp_rec.empno);
DBMS_OUTPUT.put_line(r_emp_rec.ename);
```

Using PL/SQL records to read `FETCH`ed data comes into its own when you are dealing with table and cursors with many columns.

Explicit Cursor - Fetching

You can FETCH more than one row from a cursor using the FETCH statement within a looping construct. For example:-

```
LOOP
      FETCH employees_cur INTO r_emp_rec;
END LOOP;
```

Each successive call to FETCH will return the next row from the active set. The problem with the above code is that the will never stop, it will FETCH all rows from the table and then continue issuing the FETCH statement but this will retrieve no data, so it will forever issue the FETCH without returning anything. To exit from the loop we need to check whether the last FETCH actually returned anything, we do this with the cursor attributes %FOUND and %NOTFOUND.

```
LOOP
      FETCH employees_cur INTO r_emp_rec;
      EXIT WHEN employees_cur%NOTFOUND;
END LOOP;
```

The above code will return all rows in the *active set*, then exit the loop the first time the FETCH returns no data.

Explicit Cursor - Closing

Once you have finished with the cursor, you should **ALWAYS** close it, you do this with the `CLOSE` statement:-

```
CLOSE employees_cur;
```

Once closed the cursor can be re-opened.

Explicit Cursor - Cursor Attributes

There are 4 explicit cursor attributes that can be used within your code, some we have seen already:-

Attribute	**Description**
%ROWCOUNT	The number of rows fetched
%FOUND	TRUE if the last `FETCH` returned a row, otherwise FALSE
%NOTFOUND	TRUE if the last `FETCH` did not return a row, otherwise FALSE
%ISOPEN	TRUE is the named cursor is already open, otherwise FALSE

Explicit Cursor - A Complete Example

Here is a complete example of using an explicit cursor:-

```
DECLARE
      -- Declare cursor to get employee info
      CURSOR employee_cur(p_job VARCHAR2)
      IS
            SELECT empno
            ,      ename
            FROM   emp
            WHERE  job = p_job;

      -- Declare record to hold cursor row
      r_emp_rec employees_cur%ROWTYPE;
BEGIN
      OPEN employee_cur('CLERK');

      LOOP
            -- Retrieve row from active set
            FETCH employee_cur INTO r_emp_rec;

            -- Exit loop if last fetch found nothing
            EXIT WHEN employee_cur%NOTFOUND;
      END LOOP;

      CLOSE employee_cur;

END;
```

Explicit Cursor - WHERE CURRENT OF

A special clause exists that can be used with SQL commands and cursors, this is the `WHERE CURRENT OF` clause. You use this clause in an `UPDATE` or `DELETE` statement instead of specifying column names in the `WHERE` clause, it allows you to reference the current row in the *active set*. It must be used in conjunction with the `FOR UPDATE` clause in the cursor declaration to ensure the rows are locked.

```
DECLARE
      -- Declare cursor to get employee info
      CURSOR employee_cur(p_job VARCHAR2)
      IS
            SELECT empno
            ,      ename
            FROM   emp
            WHERE  job = p_job
            FOR UPDATE;

      -- Declare record to hold cursor row
      r_emp_rec employees_cur%ROWTYPE;
BEGIN
      LOOP
            -- Retrieve row from active set
            FETCH employee_cur INTO r_emp_rec;

            UPDATE emp
            SET sal = sal * 1.15
            WHERE CURRENT OF employee_cur;

            -- Exit loop if last fetch found nothing
            EXIT WHEN employee_cur%NOTFOUND;
      END LOOP;

      CLOSE employee_cur;

END;
```

Explicit Cursor - `WHERE CURRENT OF`

The previous example retrieves and locks rows from the emp table, then the rows are updated using `WHERE CURRENT OF` clause.

The `FOR UPDATE` clause will ensure the rows are locked when the `OPEN` cursor is performed. If a lock cannot be obtained because some other process is holding locks on the same data, then the program will wait until these locks are free and the locks can all be obtained successfully. You can change this behaviour by adding the keyword `NOWAIT` after `FOR UPDATE` in the cursor declaration, this specifies that the program should not wait until locks are obtained, doing this though will not allow you to use the `WHERE CURRENT OF` clause in a SQL statement.

Explicit Cursor - Cursor FOR Loop

Within PL/SQL there is a special kind of FOR loop that is designed for working with explicit cursors. A Cursor FOR Loop OPENs, FETCHes from and CLOSEs a cursor automatically. The data from the *active set* is automatically put into a record that is implicitly created. The basic syntax for a cursor FOR loop is as follows:-

```
DECLARE
      CURSOR <name>[parameters]
      IS
            <query ….>
BEGIN
      FOR <record name> in <cursor name>[parameters]
      LOOP
            . . .
      END LOOP;
END;
```

The record name does not need to be declared as it is automatically done so for you and it is in scope for the duration of the loop. Using cursor FOR loops in a great deal easier than manually processing cursors.

Explicit Cursor - Cursor FOR Loop

Using a snippet of code for an example, we can clearly see the advantage of using cursor FOR loops.

Without Cursor FOR Loop

```
      . . .
      r_emp_rec employees_cur%ROWTYPE;
BEGIN
      OPEN employee_cur('CLERK');

      LOOP
            FETCH employee_cur INTO r_emp_rec;
            <process data here>
            EXIT WHEN employee_cur%NOTFOUND;
      END LOOP;

      CLOSE employee_cur;

END;
```

With Cursor FOR Loop

```
BEGIN
      FOR r_emp_rec IN employee_cur('CLERK')
      LOOP
            <process data here>
      END LOOP;
END;
```

Notice that there is no record declaration, OPEN, FETCH, EXIT or CLOSE statements.

Explicit Cursor - Cursor FOR Loop

You can even use cursor FOR loops without declaring the cursor first, this is called a Cursor FOR Loop with a SELECT sub-statement, it basically has the following form:-

```
BEGIN
      FOR r_emp_rec IN (SELECT ename FROM EMP)
      LOOP
            <process data here>
      END LOOP;
END;
```

As you can see, this form of cursor FOR loop does not use a declared cursor, instead the query is defined within the loop itself.

One drawback of using this type of cursor FOR loop is the fact that you cannot read cursor attributes, this is because the cursor is not declared with a name.

Summary

- Explicit Cursors are named PL/SQL constructs for querying multiple rows
- You use the `OPEN` statement to open an explicit cursor
- Rows are retrieved with the `FETCH` statement
- The `CLOSE` statement closes the cursor
- Cursor Attributes - `ROWCOUNT`, `FOUND`, `NOTFOUND` and `ISOPEN`
- `WHERE CURRENT OF` clause
- Cursor `FOR` Loops - Using declared cursors and `SELECT` sub-statements

What is Next?

The next section deals with Stored Procedures and Functions. All the code we have seen so far have been in the form of an Anonymous Block, this is an unnamed block of PL/SQL that you execute by providing SQL*Plus with the actual filename of the code, Stored Procedures and Functions are blocks of code which are stored in the database and can be executed from any other PL/SQL block.

Section Six

Stored Procedures & Functions

Stored Procedures & Functions

In this section we will learn how to create and execute our own stored procedures and functions. A stored procedure or function is a block of PL/SQL with a name that is stored in the database. This named PL/SQL block can be called from any other PL/SQL block, whether it be an anonymous block or another procedure or function. We will cover:-

- Stored Subprograms - An overview
- Procedures and Functions - The difference
- Creating a Stored Procedure
- Creating a Stored Function
- Handling Exceptions in Stored Subprograms
- Local Subprograms
- Guidelines
- Benefits of Stored Subprograms

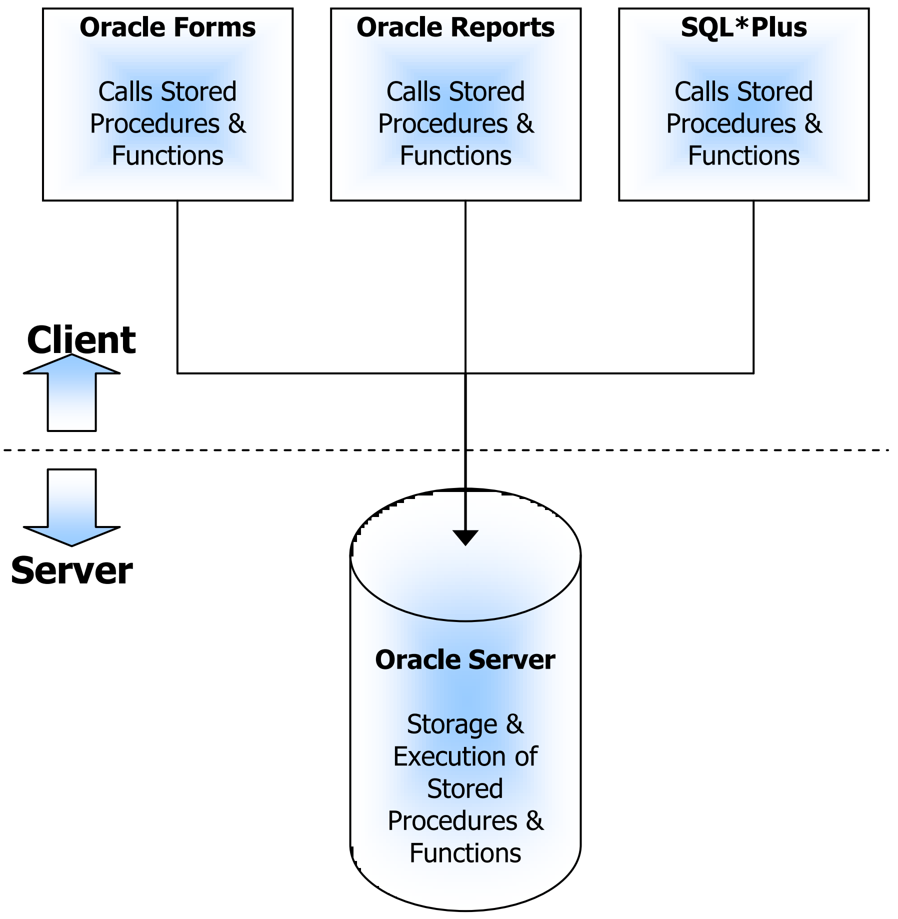
Stored Subprograms - An Overview

The collective name for stored Procedures and Functions is Stored Subprograms, this also includes two other PL/SQL constructs called Packages and Triggers, these will be covered in a later section.
A Stored Procedure or Function can be called from almost anywhere, this could be SQL*Plus, Oracle Forms & Reports or within another Stored Subprogram. Stored functions can even be executed from within a SQL statement.

The stored code is held in compiled form within the database itself, this gives it an added performance benefit over an anonymous block, which is parsed and compiled at run-time.

Stored Subprograms - An Overview

The following diagram illustrates how stored procedures are invoked, stored and executed.


Oracle Forms
Calls Stored Procedures & Functions
Oracle Reports
Calls Stored Procedures & Functions
SQL*Plus
Calls Stored Procedures & Functions
Client
Server
Oracle Server
Storage & Execution of Stored Procedures & Functions

You can partition the logic of your application very easily using stored procedures and functions. When developing an application, try to keep all logic that requires data access within the database itself, all client activity should reside in the client, this can drastically reduce network traffic.

Procedures & Functions - The Difference

So what is the difference between a procedure and a function? Basically, a procedure can be thought of as a PL/SQL or SQL command, like the `COMMIT` command for instance. `COMMIT` is a command that performs some action, it is used by itself, not as part of any other statement, whereas a function is part of an expression, a function has a value, its return value.

Oracle itself comes with many functions, `NVL`, `DECODE`, `SUBSTR`,…etc, are all functions, they usually accept data and return some data, this data is then used as part of another statement.

Procedures and functions are created in almost the same way, the major difference being that a function requires a return value. We will take a look at creating and using both procedures and functions.

Creating a Stored Procedure

You create a stored procedure with the `CREATE PROCEDURE` statement. Its has the following syntax:-

```
CREATE [OR REPLACE] PROCEDURE proc_name
      [(argument [IN|OUT|IN OUT] type [DEFAULT value]
      . . .
      argument [IN|OUT|IN OUT] type)]
IS|AS
      <declarative section>
BEGIN
      <procedure body>
[EXCEPTION
      <exception handlers>]
END;
```

Generally, always use `CREATE OR REPLACE`, this ensures that the procedure is always created, if you used `CREATE` then an error would occur if you attempted to re-create it, you would have to `DROP` the procedure first.

Creating a Stored Procedure

Procedure Name

The procedure name can be almost anything you like, use the same naming rules as you would for any other database object, in general, try to keep the name meaningful but short.

Argument List

The argument list specifies what arguments /parameters the procedure can accept. You can have as many arguments as you like. You first specify the argument name followed by the argument mode, this can be one of:-

Mode	Description
`IN`	Input parameter only, the value passed is input only. It is not returned to the invoking statement. If you omit mode, then `IN` is assumed.
`OUT`	Output only. These parameters are used to pass values back to the invoking statement
`IN OUT`	Both input and output. Values can be read from and passed back the invoking statement.

You follow the mode with the datatype of the argument, this can be one of `DATE`, `NUMBER` or `VARCHAR2`. You can if you wish anchor the datatype to a column on a table.

Creating a Stored Procedure

Declarative Section

After the argument list follows the declarative section of the procedure, this is the same as the `DECLARE` section of an anonymous block but using the `IS` or `AS` keyword. In here you put all your local variables and cursor declarations.

Procedure Body

Now comes the main part of the procedure, its body. In here goes the actual code of the procedure. This can be anything you like.

Exception Section

A procedure being just a PL/SQL block, can have its own exception section. You would handle all exceptions for the procedure in here.

Creating a Stored Procedure

Actual & Formal Parameters

When you invoke a procedure, you may provide parameters, these are referred to as *Actual Parameters*. The argument list in the procedure definition what is known as the *Formal Parameter* list. At run time, the *Actual Parameter* values are copied into the *Formal Parameters* that are then used throughout the procedure body. Any arguments specified as `OUT` or `IN OUT` are then copied from the *Formal Parameters* back into the *Actual Parameters* when the procedure ends.

Anything declared within the procedure is only within scope for the duration of the procedure and cannot be used by anything other than the procedure.

Creating a Stored Procedure

Example 1

Okay, enough talk, lets see some actual code to create a procedure.

The following code creates a simple procedure that is used to log debug information into a debug table:-

```
CREATE OR REPLACE PROCEDURE debug
                  (     p_program_name IN VARCHAR2
                  ,     p_text         IN VARCHAR2)
IS
BEGIN
      INSERT INTO debug
      (     program_name
      ,     text
      ,     logged_at ) VALUES
      (     p_program_name
      ,     p_text
      ,     SYSDATE);
END;
```

The above procedure accepts two arguments, one containing the program name you are debugging, the other is the debug text. The procedure simply inserts a row into a table called debug.

Now anywhere where you want to insert data into the debug table, you simply make a call to the procedure rather than having to write the `INSERT` statement everytime.

Creating a Stored Procedure

Invoking Example 1

This debug procedure can be invoked from within another PL/SQL block as follows:-

```
debug('My Program','Debug text is here');
```

And that's it!!

You can execute a procedure directly within SQL*Plus by entering:-

```
EXEC debug('My Program','Debug text is here');
```

The `EXEC` SQL*Plus command simply surrounds the PL/SQL code (in this case the procedure call to debug) with `BEGIN` and `END` keywords, making it an anonymous block.

Creating a Stored Procedure

Example 2

The following code creates a procedure that is used to obtain information about an employee:-

```
CREATE OR REPLACE PROCEDURE EmpInfo
                  (     p_empno      IN   emp.empno%TYPE
                  ,     p_ename      OUT  emp.ename%TYPE
                  ,     p_job        OUT  emp.job%TYPE
                  ,     p_deptno     OUT  emp.deptno%TYPE)
IS
      CURSOR emp_info_cur(p_empno emp.empno%TYPE)
      IS
            SELECT      ename
                  ,     job
                  ,     deptno
            FROM        emp
            WHERE       empno = p_empno;

      r_emp_info emp_info_cur%ROWTYPE;
BEGIN
      OPEN emp_info_cur(p_empno);

      FETCH emp_info_cur INTO r_emp_info;

      CLOSE emp_info_cur;

      p_ename := r_emp_info.ename;
      p_job := r_emp_info.job;
      p_deptno := r_emp_info.deptno;

END;
```

The above procedure takes 4 arguments, the first is the employee number you want information about, the next 3 are output parameters that will contain the information when the procedure ends.

Creating a Stored Procedure

Invoking Example 2

The following code might be used to invoke this procedure:-

```
DECLARE
      l_ename      emp.ename%TYPE;
      l_job        emp.job%TYPE;
      l_deptno     emp.deptno%TYPE;

BEGIN
      EmpInfo(     7902
            ,      l_ename
            ,      l_job
            ,      l_deptno);

      DBMS_OUTPUT.put_line(l_ename);
      DBMS_OUTPUT.put_line(l_job);
      DBMS_OUTPUT.put_line(l_deptno);

END;
```

The above code declares 3 local variables, these are used to hold the values that `EmpInfo` returns. The procedure is then called and the local variables are then displayed.

Creating a Stored Procedure

Positional & Named Parameters

Actual parameters are matched with formal parameters by their position in the argument list, for example, if we have a procedure that accepts 3 parameters, then the first formal parameter is matched with the first actual parameter when the procedure is invoked, the second with the second and so on.

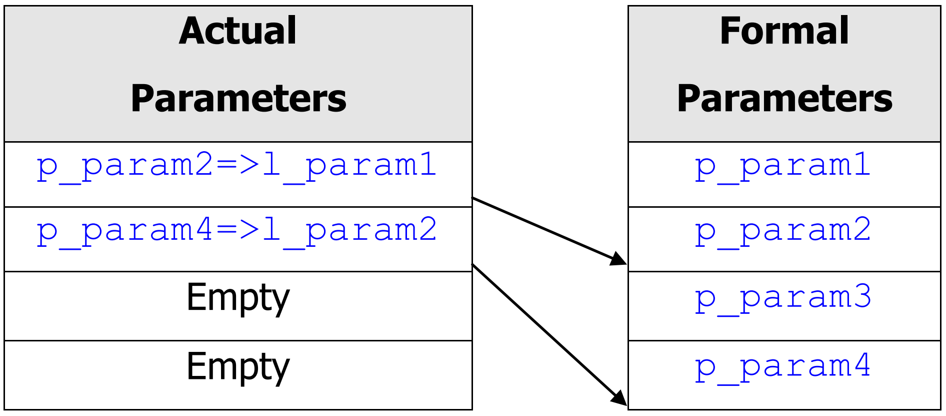
Actual Parameters		**Formal Parameters**
`l_param1`	→	`p_param1`
`l_param2`	→	`p_param2`
`l_param3`	→	`p_param3`
`l_param4`	→	`p_param4`

Each actual parameter value is copied into its positional formal parameter equivalent, wholly based on its position in the parameter list. Any parameter with default values you do not wish to specify **MUST** be at the end of the list so as not to affect the position of the parameters.

Creating a Stored Procedure

Positional & Named Parameters

Another method of providing parameters is to use Named Notation, this is where you give the actual Formal parameter names when you invoke the procedure or function. This allows you to specify any parameters in any order.



Actual Parameters
p_param2=>l_param1
p_param4=>l_param2
Empty
Empty

Formal Parameters
p_param1
p_param2
p_param3
p_param4

When invoking the procedure, you prefix each actual parameter with its formal parameter name, for example,

```
BEGIN
      SomeProc(    p_param2 => 10
            ,      p_param4 => l_string);
END;
```

Both notations can be mixed but if you do so you must ensure all positional parameters appear before any named parameters.

Creating a Stored Function

A function is created in almost the same way as a procedure, they only major difference is the fact the a function needs a return value. A procedure is a PL/SQL statement itself, whereas a function is what is known an RVALUE, this is something that can appear on the right hand side of an expression, for example:-

```
      l_full_name := GetName(p_empno);
```

The above code assigns the result of a function called `GetName` to a variable called `l_fullname`, in this case the function accepts a single parameter, the employee number and it returns the name for that employee.

Functions can be used almost anywhere within an expression, this could be the condition of an `IF` statement, a `WHILE` condition or any number of other types of expression. Functions (with some restrictions) can also be used within a SQL statement.

Creating a Stored Function

Functions are created with the `CREATE FUNCTION` statement, it has the following syntax:-

```
CREATE [OR REPLACE] FUNCTION proc_name
      [(argument [IN|OUT|IN OUT] type [DEFAULT value]
      . . .
      argument [IN|OUT|IN OUT] type)]
      RETURN return-type
IS|AS
      <declarative section>
BEGIN
      <function body>

[EXCEPTION
      <exception handlers>]
END;
```

The function body should include as least one `RETURN` statement, it has the following syntax:-

```
RETURN expression;
```

Where `expression` is any valid expression. The type of the expression is converted to the type specified by the `RETURN` clause in the function definition. The `RETURN` statement passes control immediately back to the calling program, you can have more than one `RETURN` but only one will be executed. You can also use `RETURN` in a procedure body but no value is returned, it can be used simply to return control to the calling program.

Creating a Stored Function

Example 1

The following code creates a function that will return an employee name based on the employee number given:-

```
CREATE OR REPLACE FUNCTION GetName
                        (p_empno emp.empno%TYPE)
RETURN VARCHAR
IS
      CURSOR empname_cur(p_empno emp.empno%TYPE)
      IS
            SELECT ename
            FROM   emp
            WHERE  empno = p_empno;

      r_empname empname_cur%ROWTYPE;
BEGIN
      OPEN empname_cur(p_empno);

      FETCH empname_cur INTO r_empname;

      IF empname_cur%NOTFOUND THEN
            r_empname.ename := 'UNKNOWN EMPLOYEE';
      END IF;

      CLOSE empname_cur;

      RETURN r_empname.ename;
END;
```

First, the above function declares a cursor to retrieve the employee name, the cursor is then opened, a fetch is performed, if no rows are found then the name is set to `UNKNOWN EMPLOYEE`, the cursor is closed and the name is returned to the calling program.

Creating a Stored Function

Invoking Example 1

The following code could be used to invoke the `GetName` function:-

```
DECLARE
      l_employee_name emp.ename%TYPE;

BEGIN
      l_employee_name := GetName(7902);

      DBMS_OUTPUT.put_line(l_employee_name);
END;
```

The function could also be called from a SQL statement as follows:-

```
SELECT GetName(empno) name
,      amount
FROM   bonus;
```

To allow functions to be called from within a SQL statement, it must meet certain restrictions, these restrictions are referred to as purity levels which are briefly covered in the next section on Packages.

Handling Exceptions

The way you handle errors that occur in your code is pretty much down to you, personal style or company/client guidelines may dictate your method. In any case, here are a few pointers for Exception Handling within stored subprograms:-

- Make your function or procedure as robust as possible, i.e. error trap anything that can go wrong - because it probably will!!
- Make use of the `EXCEPTION` section in your code, handle local exceptions in here and pass the exception on to the calling program using a statement called `RAISE_APPLICATION_ERROR`, see next slide for an example.
- If a procedure can either work or fail then try to convert it into a function with a `BOOLEAN` return value, this should be `TRUE` if the function worked okay, otherwise `FALSE`.
- You can use `OUT` or `IN OUT` parameters in procedures and function to pass back status information.

Handling Exceptions

RAISE_APPLICATION_ERROR

This statement can be used to pass error information back to a calling program, it takes the following syntax:-

```
RAISE_APPLICATION_ERROR(error_number,error_text);
```

Where the `error_number` is any number between -20000 and -20999 and `error_text` is the text you want to appear as the error message in the calling program. Once called, this statement immediately passes control back to the calling program and raises your error as if it were a user-defined exception, it is then for the calling program to handle the exception correctly.

NOTE - Testing and Debugging

To test and/or debug a stored subprogram, it is common practice to create a small PL/SQL block that can be executed directly within SQL*Plus. You can display debug messages directly to the screen using the `DBMS_OUTPUT` package.

Handling Exceptions

RAISE_APPLICATION_ERROR

Here is an example:-

```
CREATE OR REPLACE FUNCTION balance
                         (p_account IN NUMBER)
RETURN NUMBER
IS
      l_balance NUMBER;
BEGIN
      SELECT balance
      INTO   l_balance
      FROM   bank_account
      WHERE  account_no = p_account;

      RETURN l_balance;
EXCEPTION
      WHEN NO_DATA_FOUND THEN
            RAISE_APPLICATION_ERROR(-20500
                        ,     'Unknown Account Number');
END;
```

The above code creates a function called `balance` that returns the current balance of a specified account. If the balance cannot be found because the account does not exist then the `NO_DATA_FOUND` exception is raised in the function, to the calling program this would probably mean nothing, so an error is raised with an error number of -20500 and error text of 'Unknown Account Number'.

Handling Exceptions

RAISE_APPLICATION_ERROR

Given the function balance, here is how you could catch the error it returns:-

```
DECLARE
      l_account_balance NUMBER;

      e_unknown_account EXCEPTION;
      PRAGMA EXCEPTION_INIT(e_unknown_account,-20500);

BEGIN

      l_account_balance := balance(14956);

EXCEPTION

      WHEN e_unknown_account THEN
            DBMS_OUTPUT.put_line('Error : '||SQLERRM);

END;
```

The above code declares a variable to hold the return value of the balance function. It also declares a user-defined exception called e_unknown_account, it then ensures the error number -20500 is associated with this exception. If an error -20500 occurs then the exception handler will catch it and act accordingly.

Local Subprograms

Stored subprograms are stored within the database itself. Another type of subprogram is one that is local only to a block of PL/SQL, these subprograms are defined in the declaration section of another block of code. They are generally created for helper routines, i.e. routines that are useful only to the program they defined in and therefore should not be available to anyone else. Local subprograms are defined in almost the same way as a stored subprogram, the only difference being the you do not prefix the `FUNCTION` or `PROCEDURE` statement with `CREATE OR REPLACE`. The definition of a local subprogram should be in the declarative section of a PL/SQL block, after any other declarations (such as variables, constants, cursors,..etc)

Local Subprograms

Here is a common example of the local subprogram:-

```
DECLARE
      l_account_balance NUMBER;

      e_unknown_account EXCEPTION;
      PRAGMA EXCEPTION_INIT(e_unknown_account,-20500);

      PROCEDURE p(p_text IN VARCHAR2)
      IS
      BEGIN
            DBMS_OUTPUT.put_line(p_text);
      END;

BEGIN

      l_account_balance := balance(14956);

      p('Balance = '||TO_CHAR(l_account_balance));

EXCEPTION

      WHEN e_unknown_account THEN
            p('Error : '||SQLERRM);

END;
```

The above anonymous block contains a local subprogram called `p`, this is simply a replacement for `DBMS_OUTPUT.put_line`, this is useful because it can simply save a great deal of typing.

A Few Guidelines

Here are a few guidelines for developing stored subprograms:-

- **Generic** - if possible, try to make your subprogram as generic as possible, this way, it is likely to be re-used more often
- **Robust** - make your subprogram as robust as possible, trap anything that can go wrong
- **One Action** - try to ensure you subprogram only performs one action, by this I mean one logical action, if more than one action is needed then more than one subprogram is needed
- **Local subprograms** - Make use of local subprograms, they can make your code easier to read and debug, though be careful, overuse can have the opposite effect
- **Arguments/Parameters** - many arguments can make your subprogram flexible but also make it hard to use, try to derive as many values as you can from other values

Benefits of Stored Subprograms

Using stored subprograms gives a number of benefits:-

Performance

- PL/SQL is parsed at compile time and not runtime
- Can reduce network traffic by bundling database calls
- Re-parsing for multiple users is reduced due to shared SQL

Security and Integrity

- Other database objects (tables,…etc) can be protected and only accessed via functions/procedures
- Ensure related actions are performed together or not at all

Development

- Shared code, write once, use all - code can be called from anywhere within an application
- Debugging is simplified
- Implement changes in one place
- Can make changes online

Summary

Stored subprograms, in particular Procedures and Functions are very powerful constructs.

- Stored and executed in the database
- Procedures are PL/SQL statements
- Functions are RVALUES's that can be used in expressions
- Functions can also be invoked from a SQL statement
- Error handling with `RAISE_APPLICATION_ERROR`
- Subprograms can be local to a block

What is Next?

In the next section we take a look at Packages. Packages are one of PL/SQL's most powerful features. They allow you to group together, variables, cursors, functions and procedures into one object.

Section Seven

Packages

Packages

This section deals with one of PL/SQL's most powerful features, the Package. In particular, we cover:-

- What is a Package?
- Packages and Scope
- Creating a Package
- Overloading
- Invoking Functions from SQL Statements
- Privileges & Security
- Oracle supplied Packages

This course is a beginners guide to PL/SQL and therefore Packages (as with all other subjects) are covered very briefly, whole volumes of text have been written describing how to implement and take advantage of packages. The following notes should be enough to get you started.

What is a Package?

A Package is another type of named PL/SQL block. A Package allows you to create related objects together. A Package is stored in the database as with Stored Procedures and Functions, though a Package cannot be local to a block. A Package has two parts, a Specification (or header) and a Body, each part is stored separately within the database. The Specification of a Package contains details of what can be found in the Package, though it does not contain any of the code that implements the Package, this code is stored within the Package Body.

Basically, a package is a declarative section of code, in that anything that can appear in the declarative section of a PL/SQL block can appear in a package, this could be variables, cursor, exceptions, functions or procedures.

What is a Package?

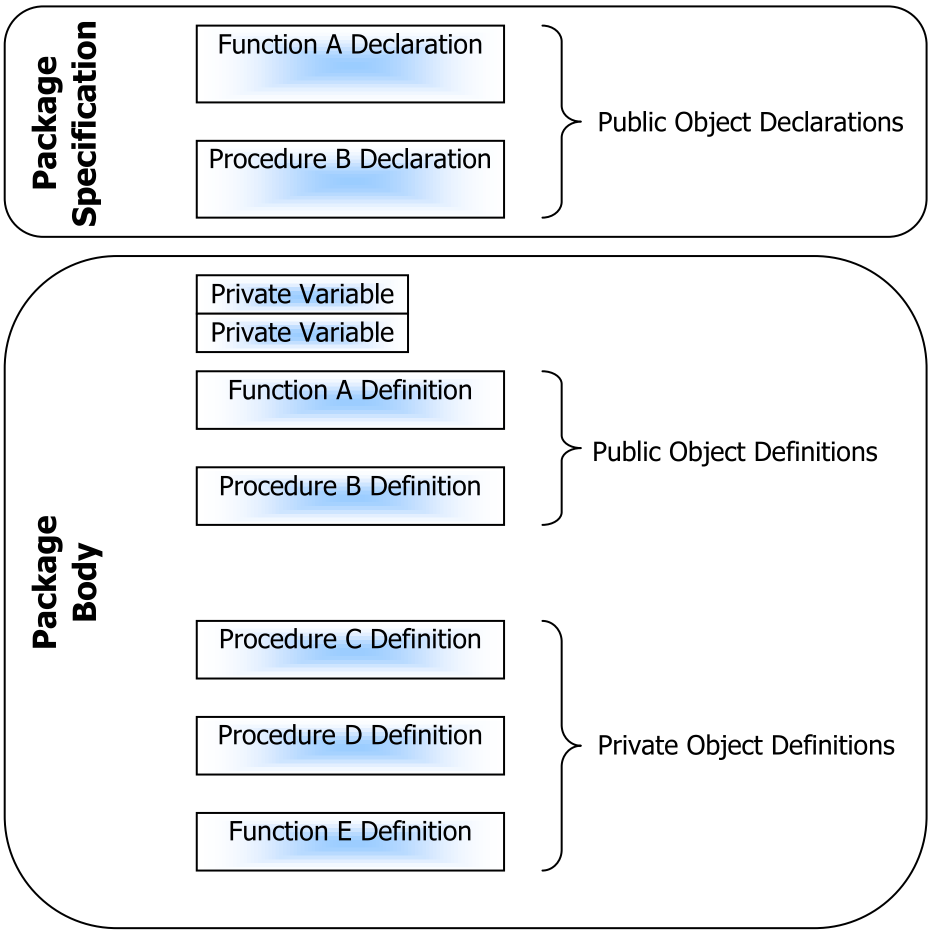
Objects declared in the specification are called public objects, that is, they can be seen and used by anyone with the correct privileges.

The package body contains the actual implementation of the package. This is the code that defines what the package actually does. Each function and procedure declared in the specification must have the definition in the body. Everything that can be declared in the specification can also be declared in the body without actually being in the specification, these objects are known as private objects, or in other words, nobody else can see or use them, they can only be seen/used by the package itself.

What is a Package?

A typical package will consist of many public functions and procedures, these are declared in the specification and defined in the body. A typical package may also contain many private variables, cursors, exceptions, functions and procedures, all defined in the body. We will look at creating a typical package over next few slides.

A Typical Package



Packages and Scope

Any object, whether it is a variable or function, that is declared in the specification is visible and within scope outside of the package so long as the object name is prefixed with package name, for example, assume we have a package called bank_account with a public function called withdraw, we can call the withdraw function from a PL/SQL block as follows:-

```
BEGIN
      BANK_ACCOUNT.withdraw(l_account_no,l_amount);
END;
```

The above function call is almost the same as calling a stored function, the only difference being the function is qualified with the owning package name.

Any object declared within the package body only, is only visible and in scope to the package itself. You may, for example, create private variables used only by the package itself, or you may create helper procedures, as in the `p` procedure created earlier.

Creating a Package

Before we write any code, we need to define what our package actually does; here is a description for our package:-

- Maintain a bank balance
- Make withdrawals (reduce balance)
- Make deposits (increase balance)
- Query balance

Also, we can define some business rules at this point:-

- Balance cannot be overdrawn
- There should be some kind of limit to how much can be withdrawn per transaction.

We now have the definition of the bank account package, now for its implementation.

Creating a Package

First, you need to create the package specification, this is done with the following statement:-

```
CREATE OR REPLACE PACKAGE package_name
IS|AS
      <package-specification>
END [package name]
```

The name can be almost anything you like, the same restrictions apply as with all identifiers. Try to make the name useful, generally the package name will relate to what its contents relate to, for example, if a package contains functions and procedures for working with bank accounts, such as `withdraw` and `deposit`, then a good package name might be `bank_account`. The package specification can be anything that would normally appear in the declarative section of a block of code (except for function and procedure bodies, this code is in the body).

The package specification and body can appear in a single file or separate files. The usual practice is to give the source filename an extension of `pkg` for a single file or `pks` (`pkh`) and `pkb` for separate files.

Creating a Package

Looking at the description for our bank account package, we can see that:-

- We require 3 public functions or procedures:-
    1. Withdrawal
    2. Deposit
    3. Balance query
- We require private data
    1. Current balance
    2. Maximum allowed per withdrawal

Using packages, we can protect our data; by making data such as balance private and providing procedures or functions to access this data, we are assured that nothing else can modify this data.

Creating a Package

Now, let's create our package specification:-

```
CREATE OR REPLACE PACKAGE bank_account
IS
      -- Procedure for making a withdrawal
      PROCEDURE withdraw(p_amount IN NUMBER);

      -- Procedure for making a deposit
      PROCEDURE deposit(p_amount IN NUMBER);

      -- Function to query the balance
      FUNCTION balance RETURN NUMBER;
END;
```

That's it!!! Our Package specification is done. This can be thought of as the public interface for our package, i.e. anything that appears in here is visible and within scope for all programs outside the package.

Keeping the package specification in a separate file facilitates modifications later, this means that you can change the body of a package (so long as the spec is not affected) without affecting any programs which use the package. It also allows you to hide your implementation.

Creating a Package

We now need to create the package body, this is where the actual code appears for all public subprograms as well as any other private objects. You create the body with the following statement:-

```
CREATE OR REPLACE PACKAGE BODY package_name
IS|AS
      <package-body>
END [package name]
```

As you can see, the only major difference for the body is the inclusion of the word `BODY` in the first line.

Creating a Package

The following code will create our package body:-

```
CREATE OR REPLACE PACKAGE BODY bank_account
IS
      --
      -- Private data objects
      --
      -- Hold account balance, initialised to 0
      v_balance                 NUMBER := 0;

      -- Hold maximum withdrawl amount
      v_max_withdrawl CONSTANT NUMBER := 250;

      --
      -- Private functions/procedures
      --
      -- Print text to screen
      PROCEDURE p(p_text IN VARCHAR2)
      IS
      BEGIN
            DBMS_OUTPUT.put_line(p_text);
      END;

      --
      -- Public function/procedure definitions
      --
      -- Procedure for making a withdrawal
      PROCEDURE withdraw(p_amount IN NUMBER)
      IS
            l_new_balance NUMBER:= v_balance - p_amount;
      BEGIN
            IF p_amount > l_max_withdrawal THEN
                  -- Ensure amount is within limit
                  p('Withdrawals limited to '||
                        TO_CHAR(l_max_withdrawl)||
                        ' per transaction');
            -- No overdrafts allowed
            ELSIF l_new_balance < 0 THEN
                  -- No overdrafts allowed
                  p('No overdraft available');
                  p('Cash available : '
                        ||TO_CHAR(v_balance));
            ELSE
                  v_balance := v_balance - p_amount;
                  p('Here is your cash!');
            END IF;
      END withdraw;
```

continued . . .

Creating a Package

```
        -- Procedure for making a deposit
        PROCEDURE deposit(p_amount IN NUMBER)
        IS
        BEGIN
              IF p_amount <= 0 THEN
                    p('Deposit must be at least £1');
              ELSE
                    v_balance := v_balance + p_amount;
                    p('Thankyou!');
              END IF;
        END;

        -- Function to query the balance
        FUNCTION balance RETURN NUMBER
        IS
        BEGIN
              RETURN v_balance;
        END;
  END;
```

To make use of this package, you might do the following:-

```
  BEGIN
        DBMS_OUTPUT.put_line(BANK_ACCOUNT.balance);

        BANK_ACCOUNT.withdraw(100);

        BANK_ACCOUNT.deposit(500);

        DBMS_OUTPUT.put_line(BANK_ACCOUNT.balance);

        BANK_ACCOUNT.withdraw(275);

        DBMS_OUTPUT.put_line(BANK_ACCOUNT.balance);
  END;
```

As you can see, the package is a very powerful construct, it lends itself to very structured code and some of the benefits of object orientation (there is small discussion on this subject later).

Overloading Subprograms

Overloading is method of creating more than one subprogram (function or procedure) with the same name but with different arguments. This allows you to create very useful subprograms that can act in different ways depending on what data they are given. Oracle comes with many overload subprograms, take the built-in function `TO_CHAR`, this function is overload even though it appears to be a single function whenever you use it, have a look at the function declarations below:-

```
TO_CHAR(number)
TO_CHAR(number,format)
TO_CHAR(date)
TO_CHAR(date,formart)
```

As you can see, the `TO_CHAR` function works in 4 different ways, with numbers and dates, with and without a format mask, whenever you use the function, the system determines which function to call based on the supplied arguments.

Overloading Subprograms

How might we use subprogram overloading in our bank account package?

Lets say we want to add functionality that allows the user to withdraw all remaining funds, we could do this in a number of ways:-

1. Get current balance and withdraw that amount
2. Change the `withdraw` procedure to add a flag which indicates you want all the cash
3. Create a new procedure to withdraw all the cash

All of the above are very possible (and easy in this case) but a much better method would be to overload the `withdraw` procedure; change it so that if no arguments are provided then withdraw all current funds.

Overloading Subprograms

Our specification would now contain two declarations of `withdraw`:-

```
PROCEDURE withdraw;
PROCEDURE withdraw(p_amount IN NUMBER);
```

The body would contain the new version of withdraw as follows:-

```
PROCEDURE withdraw
IS
BEGIN
      v_balance := 0;
END;
```

or even . . .

```
PROCEDURE withdraw
IS
      withdraw(balance);
END;
```

The second method is better still, as it makes use of the existing `withdraw` function which is good because additional functionality does not have to be repeated, it also makes use of the `balance` function to find out the current balance. So `withdraw` without any arguments is just a wrapper for `withdraw` with arguments. To the user, it's just a single function.

Invoking Packaged Functions from SQL Statements

As mentioned earlier, stored functions can be executed from within a SQL statement, examples of these are `TO_CHAR`, `NVL`, `DECODE`, `SUBSTR`,…etc.

You can create your own functions that can be invoked this way so long as your function meets certain restrictions. These restrictions are determined by Purity Levels. The Purity Levels tells PL/SQL what kind of things your function does. The Purity level is automatically determined for stored functions but for packaged functions you must manually set the purity level, this is because when compiling code with a packaged function call, the compiler is only concerned with the package specification when determining if your code is correct. As you know, the actual code for a package is in the body and this is not used at compile time, so purity levels cannot be determined.

Invoking Stored Functions from SQL Statements

Purity Levels

There are four Purity Levels for functions:-

Purity Level	Meaning	Description
`WNDS`	Writes no database state	The function does not modify ANY database table using DML.
`RNDS`	Reads no database state	The function does not read ANY database tables using SELECT
`WNPS`	Write no package state	The function does not modify any packaged variables
`RNPS`	Reads no package state	The functions does not read any packaged variables

Using the above Purity Levels, the following restrictions are applied:-

- Any function that is called from a SQL statement **CANNOT** modify database tables (`WNDS`)
- To allow functions to be executed remotely or in parallel, a function **CANNOT** read or write packaged variables (`WNPS`,`RNPS`)
- Functions called within `SELECT`, `SET` or `VALUES` clauses can write packaged variables, all others must have `WNPS`

Invoking Stored Functions from SQL Statements

In addition to the restrictions already mentioned, the following must also be taken into account:-

- The function **MUST** be stored in the database, local subprograms cannot be used
- Only parameters with a mode of `IN` can be used.
- The return type and formal parameters of the function **MUST** be a valid database type (i.e. not `BOOLEAN`, `RECORD`,..etc)

NOTE - Testing and Debugging

To test and/or debug a packaged procedure or function, it is common practice to create a small PL/SQL block that can be executed directly from the SQL*Plus prompt, you can display debug messages directly to the screen using the `DBMS_OUTPUT` package. Use the `SYSTEM.dual` table for testing packaged functions within a SQL statement.

Invoking Stored Functions from SQL Statements

So, how do we manually set the purity level of a packaged function?

`RESTRICT_REFERENCES` Pragma

You need to inform the compiler of the purity level of the function within the package specification with a compiler pragma. This pragma has the following syntax:-

```
PRAGMA RESTRICT_REFERENCES(function, purity...);
```

For example, to allow our `BANK_ACCOUNT.balance` function to be used within SQL we would need to first determine the purity levels then apply them. We can see that our function simply reads a packaged variable, so the only purity level not applicable is RNPS (read no package state), therefore:-

```
PRAGMA RESTRICT_REFERENCES(balance,WNPS,RNDS,WNPS);
```

would need to appear directly after the function declaration in the package specification.

Privileges & Security

What follows is a few notes on Oracle privileges and security and why they are important to PL/SQL.

Before you can create and/or execute any kind of stored program, you need the correct privileges to do so. This is usually controlled by your DBA but you should still be aware of it as there maybe occasions where you have created a stored subprogram and you need to allow other people to execute your code.

Privileges & Security

The following table lists the operations you may want to perform and the required privileges to do so.

Operation	**Required Privilege**	**And . . .**
CREATE	`CREATE PROCEDURE` system privilege	Access to all objects referenced by the procedure
CREATE OR REPLACE	Ownership of the procedure and `CREATE PROCEDURE` system privilege or `CREATE ANY PROCEDURE` system privilege	Access to all objects referenced by the procedure
ALTER	Ownership of procedure or `ALTER ANY PROCEDURE` system privilege	Access to all objects referenced by the procedure
DROP	Ownership of procedure or `DROP ANY PROCEDURE` system privilege	
EXECUTE	Ownership of procedure or `EXECUTE` object privilege or `EXECUTE ANY PROCEDURE` system privilege	

Stored subprograms execute in there owners security domain, this means you can allow other users indirect access to objects that a subprogram references so long as that user has `EXECUTE` privilege on your procedure.

Oracle Provided Packages

Oracle comes with many pre-built packages (both client and server), we have already seen `DBMS_OUTPUT`, this is simply a package to add functionality to PL/SQL for input and output. There are many other packages available.

Going through each of the supplied packages is beyond the scope of this course, as we are only concerned with learning the basics for now, but I will mention them so that you are aware of what is available (server packages only). You can then take it upon yourself to further investigate the functionality and use of these packages.

Oracle Provided Packages

The following tables lists some of the supplied packages available:-

Supplied Package	**Functionality**
DBMS_ALERT	Signal that an event has occurred within the database
DBMS_APPLICATION_INFO	Allows registration of an application with the database, used for tuning purposes.
DBMS_DDL	Compile procedures, functions and packages
DBMS_DESCRIBE	Helps obtain information about database objects
DBMS_JOB	Schedule periodic jobs within the database
DBMS_LOCK	Performs application lock synchronisation
DBMS_OUTPUT	General input/output package
DBMS_PIPE	Allows programmatic communication between database sessions
DBMS_SQL	Allows the creation of dynamic SQL & PL/SQL within PL/SQL
DBMS_TRANSACTION	Control of logical transactions
UTL_FILE	Host file input and output

Many other packages exists:-

DBMS_AQ, DBMS_AQADM, DBMS_DEFER, DBMS_DEFER_SYS, DBMS_DEFER_QUERY, DBMS_LOB, DBMS_REFRESH, DBMS_SNAPSHOT, DBMS_REPCAT, DBMS_REPCAT_AUTH, DBMS_REPCAT_ADMIN, DBMS_ROWID, DBMS_SESSION and DBMS_SHARED_POOL.
…and many more.

Summary

As we have seen, the Package is a very powerful construct.

- A Package is a construct used for grouping related data and functions/procedures
- A Package consists of a specification and a body
  - Public object declaration in specification
  - Private and public object definition in body
- Packages are created with the `CREATE PACKAGE` statement
- Function/Procedure Overloading
- Packaged Functions can be invoked from with SQL Statements though with some restrictions
- Oracle provides many pre-built packages

What is Next?

The next and final section deals with the remaining type of named subprogram, the database Trigger. A database Trigger is a PL/SQL block that is implicitly executed when a particular event in the database occurs.

Section Eight

Triggers

Triggers

We now come to the remaining type of named subprogram, the database Trigger. In this section we cover:-

- What is a Trigger?
- Trigger types
- Creating a Trigger
- Restrictions on Triggers

This course is a beginners guide to PL/SQL and therefore Triggers (as with all other subjects) are covered very briefly. The following notes should be enough to get you started.

What is a Trigger?

A trigger is a block of PL/SQL which is implicitly invoked (fired) whenever a particular event occurs in the database.

The PL/SQL within a trigger can be almost anything that would appear in a normal block with a few restrictions.

Whenever any data on a database table changes, via a DML statement (`INSERT`, `UPDATE` or `DELETE`) this is considered an event, triggers can be attached to these events.

Triggers are very much like any other named PL/SQL block in that they have a name, a declarative section, a body and an exception section, though triggers cannot accept any arguments, nor can they be fired explicitly.

What is a Trigger?

Triggers have many possible uses, they could for example, be used for:-

- Auditing System - because triggers are guaranteed to fire, they are an ideal method of auditing changes to a table, for example, you could audit/log changes to a customer record.
- Automatically signal other programs that they need to perform some action
- Archiving System - Archiving data can be achieved very easily with triggers
- Maintaining Derived Values - Derived values can be maintained with ease, for example, a stock system need never actually change the stock quantity, a trigger on the transaction table could maintain it for you
- Complex Integrity Constraints - You may have some complex integrity constraints that cannot be implemented as declarative constraints on the table

Trigger Types

The trigger type determines when the trigger actually fires. There are 12 trigger types in all, these types fall into one of two categories, the type category:-

- Statement - These triggers fire in response to a DML statement being issued
- Row - These triggers fire for all rows affected by a DML statement. These are identified by the `FOR EACH ROW` clause in the trigger definition.

Each of the above types has a further six types, these belong, again, to one of two categories, the timing category:-

- `BEFORE` - These triggers fire before the statement is executed
- `AFTER` - These fire after the statement has executed

Now the event category:-

- `INSERT`, `UPDATE` & `DELETE` - This determines the kind of DML statement that fires the trigger.

Creating a Trigger

Triggers are created with the `CREATE TRIGGER` statement, it has the following syntax:-

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE|AFTER trigger_event [OR trigger_event ...]
[OF column_list ] ON table
[FOR EACH ROW]
[WHEN (trigger condition)]
BEGIN
      <trigger body>
END;
```

Trigger Name

This can be any legal identifier, again, as with all objects, it is a good idea the make the name meaningful, so for example, a trigger to audit an items price might be called, `audit_item_price`.

Trigger Event

This is the keyword `INSERT`, `UPDATE` or `DELETE`. Multiple events can be specified using the `OR` keyword, such as `INSERT OR UPDATE`.

Creating a Trigger

Column List

This specifies the columns you want checking before the trigger fires, the correct event may have occurred but if the column list does not include a column that was changed, the trigger will not fire.

Table

This specifies the triggering table.

`FOR EACH ROW` Clause

If present, then the trigger will file for each row affected by the DML statement.

Trigger Condition

This acts as a condition that can be applied to determine whether the trigger should fire or not. You add a condition with the `WHEN` clause, the condition itself can be practically anything that would normally appear in an `IF` statement. Only valid in a **ROW** level trigger. Enclose the whole condition is parentheses.

Trigger Body

This is the actual code of the trigger. It is usual for this simply to contain a stored procedure call.

Creating a Trigger

As an example, we shall create a trigger that maintains a stock record based on a transactions table. The definition of our trigger is:-

- We have a transaction table that records issues and receipts
- We have a stock table that contains data relating to current stock figures
- We want to automatically maintain the current stock quantity based on **ONLY** transactions of type ISS (issue) and RCT (receipt).

NOTE - Testing & Debugging

Testing and Debugging Triggers is a little tricky, this is because you cannot output anything to the screen from a trigger, you have to log execution in another table or an external file. To actually fire the trigger you have to cause the triggering event to occur, this is usually done with DML entered directly at the SQL*Plus prompt.

Creating a Trigger

To create a trigger to accomplish the above, the following code is needed:-

```
CREATE OR REPLACE TRIGGER stock_quantity
      BEFORE INSERT
      ON transactions
      FOR EACH ROW
      WHEN new.transaction_type IN ('ISS','RCT')
DECLARE
      l_use_quantity stock.quantity%TYPE;

BEGIN
      IF :new.transaction_type = 'ISS' THEN
            l_use_quantity := :new.quantity * -1;
      ELSE
            l_use_quantity := :new_quantity;
      END IF;

      UPDATE stock
      SET    quantity = quantity + l_use_quantity
      WHERE  item = :new.item;
END;
```

The above trigger will fire before any rows are inserted into the transactions table, only when the transaction type is either ISS (issue) or RCT (receipt). If the transaction is an issue then the quantity is made negative to ensure stock is reduced. The quantity on the stock table is then updated by the transaction quantity for the transaction item.

Notice the use of the `new` keyword, we will discuss this next.

Creating a Trigger

You can reference values on the triggering table during execution of the trigger with the new and old keywords.

The new keyword is used to reference the value of columns on a new or updated row, whereas the old keyword is used to reference the value of a column on a row being deleted or updated. See the following table:-

Operation	OLD Value	NEW Value
INSERT	NULL	the new inserted value
UPDATE	the column value before the update	the column value after the update
DELETE	the column value before deletion	NULL

Within the body of the trigger, you need to prefix the old and new keywords with a colon (:), for example,

```
:new.value = another_value;
```

Within the WHEN clause, **DO NOT** prefix the keyword with a colon.

Creating a Trigger

You may want to combine trigger events into one trigger, i.e. your trigger may fire on INSERT OR DELETE, but you may still want to take different actions depending on the actual triggering event. You can do this by using the INSERTING, UPDATING and DELETING functions.

These functions simply return TRUE or FALSE depending on what the trigger event was. Here is an example:-

```
CREATE OR REPLACE TRIGGER audit
      BEFORE INSERT OR UPDATE OR DELETE on items
      FOR EACH ROW
BEGIN
      IF INSERTING THEN
            INSERT INTO audit_log('Row Inserted');
      ELSIF UPDATING THEN
            INSERT INTO audit_log('Row Updated);
      ELSIF DELETING
            INSERT INTO audit_log('Row Deleted);
      END IF;
END;
```

The above trigger will simply insert a row into the audit_log table, the text inserted depends on what the trigger event is.

Restrictions on Triggers

There are a number of restrictions placed on the code that is placed within a trigger body, these restrictions also apply to any stored subprogram invoked from the trigger.

- You **CANNOT** use `COMMIT`, `SAVEPOINT` and `ROLLBACK` statements.
- Do not change data in the primary key, foreign key or unique key columns of a constraining table including the triggering table. A constraining table is a table that may need to be queried for referential integrity purposes. This can create what is called a mutating table. Solving problems with mutating tables is a complex topic and is beyond the scope of this course.
- Do not read values from the changing table, use `:old` and `:new` instead.
- Be aware of the performance implications, i.e. a table with 10 triggers firing on `INSERT`'s will be much slower than inserting into a table with no triggers.

Summary

- Triggers are named PL/SQL blocks and have many possible uses
- Triggers fire when an event on the database occurs
- There are 12 trigger types, Row level, Statement Level, `BEFORE` and `AFTER`, `INSERT`, `UPDATE` and `DELETE`
- Create triggers with the `CREATE TRIGGER` statement
- Restrict firing with `WHEN` clause
- Reference triggering table columns with `:old` and `:new`
- Using `INSERTING`, `UPDATING` and `DELETING` functions when combining trigger events
- Be aware of restrictions and possible performance issues

What is Next?

We have reached the end of the course, if you followed all sections and attempted the exercises, you should be able to produce your own PL/SQL programs. You should also be able to support some existing PL/SQL programs, in that you will be able to read and understand most of the code.
We have covered a great deal of topics in a very short space of time so do not be too concerned if you haven't quite taken it all in, as said throughout the course, it is a beginners guide, it is meant to equip you with enough knowledge to at least get you started on the road to becoming a PL/SQL Developer.

If you can, starting working with PL/SQL as soon as possible, re-read these course notes and any other documentation you can find. Your real PL/SQL knowledge with come from experience, the more you get the better you will be.

Good Luck!!