# LO9 – Embed non-procedural queries in a procedural language

## 9.1 – Use Oracle’s PL/SQL syntax for decision and looping structures.

## PL/SQL Basics

### What is PL/SQL?

* Stands for Procedural Language Extension and it is specific to Oracle. (Example: Postgres uses pl/pgsql, Microsoft uses T-SQL for SQL Server)
* Allows one to use procedural programming along with SQL to accomplish tasks.

### Why use PL/SQL?

* Transactions should be in a procedure or a function. DBMS can be setup to auto commit every statement. This is why transactions (commits / rollbacks) should be in a procedure/function, otherwise each statement would be committed and you wouldn’t be able rollback.
* LAYERS – Keep your programming layers separate. If you’re interacting with Oracle, this is your **data layer** when creating your applications.
* It is faster for SQL server to handle all of the processing then it to be passing the data back and forth between the SQL server and the computer running your program.

### Advantages of PL/SQL

* Can store code on DBMS for use/reuse at a later time
* Procedural means we can use loops and conditions
* Can improve the performance because the server will process the statements as a single block which reduces network traffic
  + Example: Crystal Reports and the thick client. About 20 queries were being run that would return about 1 million records each. Would take at least 30 minutes to run the report.
* Provides error handling – one can catch errors and perform actions like displaying a message to the user.

### PL/SQL Block

* Component that makes up a PL/SQL program
* Is composed of SQL and PL/SQL statements
* Can use data manipulation language (DML) (select, insert, update, delete)
* Can use data definition language (DDL) (create, alter, drop)

A block has 3 sections:

1. **Declaration**
   1. Optional
   2. Declares place holders (variables, constants, records, cursors)
2. **Execution**
   1. Required
   2. Where the program logic is written (loops, conditionals, sql)
3. **Exception (Handling)**
   1. Optional
   2. Errors are handled by this section
   3. Errors not handled will cause the block to terminate abruptly with errors.

So at the lowest level, a block would look like:

DECLARE

*(Variable Declaration Area)*

BEGIN

*(Program Execution area)*

EXCEPTION

*(Exception/Error handling area)*

END;

NOTE: you can nest blocks within another block. These inner blocks are contained within the execution section of the outer blocks.

**Example of nested blocks:**

DECLARE

BEGIN

DECLARE

BEGIN

Outer block

Inner block

EXCEPTION

END;

EXCEPTION

END;

#### Variable names

* Must begin with a letter
* Can have up to 30 characters
* letters, numbers, $, #, \_ are all fair game to use in a variable name

#### Available datatypes

<https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/datatypes.htm#i10924>

**SQL datatypes**

* + VARCHAR2(size), CHAR(size) Sizes can be from 1-32767
  + NUMBER (size, decimals)
  + DATE

**PL/SQL datatypes**

* + BINARY\_INTEGER
    - Stores values in binary format
    - Has slightly smaller storage space requirements
    - Used for signed integers
  + BOOLEAN
    - Has a null value until it is assigned TRUE or FALSE

#### Variable Declaration

* Each variable declaration is a separate statement that ends with a semi colon

VariableName datatype [NOT NULL := value];

VariableName – name of the variable

Datatype – valid pl/sql or sql datatypes

NOT NULL – optional specification. NOTE: if a value is specified as NOT NULL, you must initialize the value when declared.

Value – optional. Used to initialize the variable

Example:

DECLARE

Salary NUMBER(4);

Dept VARCHAR2(10) NOT NULL:= “HR Dept”;

BEGIN…

#### Variable Scope

* Since blocks can be nested, any variable accessible to the outer block is **also** available to the inner blocks
* Variables in outer blocks act similar to a global variable.
* Variables declared in an inner block are **not** available to any outer blocks.
* Variables declared in an inner block act similar to a local variable

**EXAMPLE of scope**

(1)DECLARE

myVar1 number(5);

BEGIN

(2)DECLARE

myVar2 number(5);

BEGIN

END;

(3)DECLARE  
 myVar3 number(5);

BEGIN

END;

END;

In the example above:

myVar1 can be accessed in all the execution areas

myVar2 can only be accessed in #2

myVar3 can only be accessed in #3

### Constants

* A value that remains unchanged throughout its use in a pl/sql block.
* Used for declaring a value and then using it, for example, for calculations
* Values must be assigned to a constant at the time of declaration, otherwise, if you try to do it in the execution sections, you will get an error.
* **CONSTANT** is a reserved word.

ConstantName CONSTANT datatype(size):=value;

Example:

DECLARE

Salary\_increase CONSTANT NUMBER(3):= 100;

BEGIN…

### Operators

1. **Arithmetic**

Exponentiation \*\*

Multiplication \*

Division /

Addition +

Subtraction –

Negation – (eg. -7)

1. **Relational**

Equal =

Not equal != or <>

Greater than >

Greater than or equal >=

Less than <

Less than or equal <=

1. **Logical**

AND

OR

NOT

1. **OTHER**

Can use other Oracle functions:

UPPER, LOWER, INSTR, SUBSTR, RTRIM, LENGTH, TO\_DATE, TO\_CHAR, TO\_NUMBER

### To obtain input from the keyboard (user)

use an ampersand (&) to tell oracle that a particular value should be input via the keyboard.

Example:

department\_id = &department\_id;

vNumber := To\_Number(‘&userinputNumber’);

### Displaying output

In SQL Developer:

1. Go to view menu
2. Click DBMS Output
3. In DBMS output panel, click the green “+” and select the connection

In pane where you type commands issue the following at the beginning of each session:

SET SERVEROUTPUT ON

This will allow your script to output to display in the script output pane.

To actually print out text from your scripts use:

DBMS\_OUTPUT.PUT\_LINE(‘string’);

Example:

DBMS\_OUTPUT.PUT\_LINE(‘Hello World’);

NOTE: any messages sent using DBMS\_OUTPUT are not sent until the sending program completes. There is no mechanism to flush output during the execution of a procedure.

### Conditional Statements

#### Ifs

1. **BASIC IF**

IF condition THEN

Statement1;

ELSE

Statement2;

END IF;

1. **ELSE IF (ELSIF)**

IF condition THEN

Statement1;

Statement2;

ELSIF condition2 THEN

Statement3;

ELSE

Statement4;

END IF;

1. **Nested if**

IF condition THEN

IF condition2 THEN

Statement1;

END IF

ELSIF condition3 THEN

Statement2;

ELSE

Statement3;

END IF;

#### Case

CASE (operand)

WHEN (value or search condition) THEN

Statement(s);

WHEN (….more )THEN

ELSE

Statements;

END CASE;

### Looping/Iterative Statements

1. Simple loop – Used for executing a number of statements at least once.

LOOP

Statements; \*DO some work

IF condition THEN \*\*The exit condition

EXIT;

END IF;

Statements; \*if doesn’t exit, do some more work

END LOOP;

1. While loop – Executes statements as long as the condition is true.  
   Remember to:
   1. Initialize variable before the loop body
   2. Increment the variable in the loop
   3. You can use EXIT or EXIT WHEN in the while loop but it’s not recommended

WHILE condition LOOP

Statements;

END LOOP;

1. For loop – Executes statements for a predetermined number of times. Iteration is between the start and end integers. It is automatically incremented by 1. Exits when the counter = the end integer.

FOR counterVar IN startInt..endInt LOOP

Statements;

END LOOP;

## PL/SQL Records

### What are records?

Records are:

* Another datatype which can act as a placeholder.
* Composite datatypes, (a combination of different scalar datatypes like char, varchar, number, etc), are allowed. It is like a row of data from a table where each scalar data type in the record holds a value.
* Technically, a record is a user-defined datatype.

### Declaring a record:

To declare a record, you must first define a composite datatype; then declare a variable of that type.

The General Syntax to define a record is:

TYPE record\_type\_name IS RECORD

(first\_col\_name column\_datatype,   
second\_col\_name column\_datatype, ...);

Replace above with:

* *record\_type\_name* – it is the name of the record
* *first\_col\_name, second\_col\_name, etc.,- the names of the fields within the record*
* *column\_datatype* – defines the datatype of the fields.

However, if you are going to base your record (of only one field) on a table already existing in your database, look below.

To base your record on a table existing in your database, use:  
*col\_name* table\_name.column\_name%type;

By declaring the field datatype in the above method, the datatype of the column is dynamically applied to the field.  This method is useful when you are altering the column specification of the table, because you do not need to change the code again.

**NOTE:** You can use also*%type* to declare variables and constants. 

The General Syntax to declare a record of a user-defined datatype (or in our case a record) is:

record\_name record\_type\_name;

Example: create a record datatype called employee\_type and declare a new variable called *employee\_rec* of type: employee\_type.

STEP 1: define our datatype (our record datatype)

STEP 2: Make a variable of that datatype.

DECLARE

TYPE employee\_type IS RECORD

(employee\_id number(5),

 employee\_first\_name varchar2(25), **\*regular declaration**

 employee\_last\_name employee.last\_name%type, **\*based on a column from table employee called last\_name**

 employee\_dept employee.dept%type);

 employee\_salary employee.salary%type;

employee\_rec employee\_type; \*Now we are making a new record called employee\_rec.

If all the fields of a record are going to be based on the columns of a table, we can declare the record as follows:

record\_name table\_name**%ROWTYPE**;

For example, the above declaration of employee\_rec can as follows:

DECLARE

 employee\_rec employee%ROWTYPE;

### Advantages of declaring the record as a ROWTYPE

1)  You do not need to explicitly declare variables for all the columns in a table.  
2) If we alter the column specification in the database table, we do not need to update the code to reflect those changes

**Disadvantage of declaring the record as a ROWTYPE:**1) Creating a record as a ROWTYPE means that fields will be created for **all** the columns in the table. This means memory will be used to create the datatype for all the fields regardless of whether they are used or not.

### Passing Values To and From a Record

**NOTE:** When you are creating a record, you are just creating a datatype, similar to creating a variable. We still need to assign values to the record and use them.

To assign values to a record you will need to assign values to the fields within the record.   
record\_name.col\_name := value;

You can also assign values to records using SELECT Statements as shown:

SELECT col1, col2

INTO record\_name.col\_name1, record\_name.col\_name2

FROM table\_name   
[WHERE clause];

If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each column separately. SELECT all the columns from the table into the record:

SELECT \* INTO record\_name

FROM table\_name   
[WHERE clause];

To retrieve a value from a field in the record and place it into another variable is:

var\_name := record\_name.col\_name;

## Cursors

Cursor:

* Is a temporary work area created in system memory when a SQL statement is executed.
* Contains information on a select statement and the rows of data accessed by it.
* This temporary work area is used to store the data retrieved from the database, so that we can manipulate this data.
* A cursor can hold more than one row, but can only process one row at a time.
* The set of rows the cursor holds is called an active set.

There are two types of cursors in PL/SQL:

1. Implicit
2. Explicit

### Implicit cursors

* Created by default when DML statements like INSERT, UPDATE, DELETE are executed. When you execute DML statements, implicit cursors are used to process these statements.
* Created when a SELECT statement that returns just one row is executed
* The user has no control over an implicit cursor.

Oracle provides implicit cursor attributes for checking the status of DML operations. The cursor attributes for checking status are %FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN.

For example, when you execute INSERT, UPDATE, or DELETE statements, the cursor attributes tell us whether any rows are affected and how many have been affected.

When a SELECT … INTO statement is executed in the PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

The status of the cursor for each of these attributes are defined in the below table.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Return Value** | **How to use** |
| %FOUND | The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row. | SQL%FOUND |
| The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row. |
| %NOTFOUND | The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ….INTO statement return at least one row. | SQL%NOTFOUND |
| The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row. |
| %ROWCOUNT | Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT | SQL%ROWCOUNT |
| %ISOPEN | TRUE, if the cursor is already open in the program, False if it isn’t | SQL%ISOPEN |

### Explicit Cursors

* Created when you are executing a SELECT statement that returns **more than one row**.
* Store multiple records, but can only process one record at a time.
* The current row is the row that cursor is currently on. Each time you fetch a row, the current row position moves to the next row.

Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

An **explicit cursor** is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

**Four steps in using an Explicit Cursor.**

1. DECLARE the cursor in the declaration section
2. OPEN the cursor in the Execution section
3. FETCH the data from cursor into PL/SQL variables or records in the Execution section
4. CLOSE the cursor in the Execution Section before you end the PL/SQL block.

1) DECLARE

Declaring a cursor:

*DECLARE*

*CURSOR cursor\_name (parameter list)IS*

*select statement*

Replace above with:

Cursor\_name = the name for the new cursor

select statement = any valid SQL SELECT statement. You can have variables inserted into your SELECT statement BUT the variables must have been declared before the cursor has been declared.

Parameter list (*optional*) = the list of variables that you want to use in your Select statement. This list is comma-separated.

*DECLARE*

*CURSOR emp\_cur IS*

*SELECT \**

*FROM emp\_tbl*

*WHERE salary > 5000;*

In the above example we are creating a cursor ‘emp\_cur’ on a query which returns the records of all the employees with salary greater than 5000. Here ‘emp\_tbl’ in the table which contains records of all the employees.

*DECLARE*

*CURSOR emp\_cur (salary\_in NUMBER)IS*

*SELECT \**

*FROM emp\_tbl*

*WHERE salary > (salary\_in);*

2) OPEN

Opening a Cursor

When a cursor is opened, the data is loaded into the cursor and the first row becomes the current row. BUT, no values have been retrieved yet.

Syntax:  
*OPEN cursor\_name; --no parameters*

*OPEN cursor\_name (parameter values); --passing in parameters*

NOTE: You have data, but the cursor has no idea which row of the data you want to access.

1. FETCH

It’s not until this step that we can actually start using our data.

Fetch the next row of data from the cursor to make it the active set.

Fetch works using the following steps:

1. When data is fetched, it is copied to the record or variables
2. The logical pointer moves to the next row
3. This row is now the current row. On every fetch statement, the pointer moves to the next row.

If you try to fetch after the last row, the program will throw an error. Use loops and explicit cursor attributes to fetch all the records when there are more than one.

Points to remember while fetching a row:

* Fetch the rows from a cursor into the PL/SQL Record or a list of variables created in the PL/SQL block.
* If you are fetching into a PL/SQL Record, the record should have the same structure as the cursor
* If you are fetching a cursor to a list of variables, the variables should be listed in the same order in the fetch statement as the columns are present in the cursor.

Syntax:

*FETCH cursor\_name INTO record\_name;*

OR   
*FETCH cursor\_name INTO variable\_list;*

4) CLOSE

Close the cursor

Tells the database that the resources associated with the cursor can be released. *CLOSE cursor\_name;*

So Put it all together to use your explicit cursor like:

*DECLARE*

*variables;*

*records;*

*create a cursor;*

*BEGIN*

*OPEN cursor;*

*FETCH cursor;*

*process the records;*

*CLOSE cursor;*

*END;*

#### Explicit Cursor Attributes

Just like our implicit cursors which have Attributes, so do our explicit

* Control the data processing while using cursors
* Help avoid errors while accessing cursors through OPEN, FETCH, and CLOSE statements.

Errors can occur when:

a) trying to open a cursor not closed in the previous operation.  
b) trying to fetch a cursor after the last operation

Available attributes for checking the status of an explicit cursor.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Return values** | **Example** |
| %FOUND | TRUE, if fetch statement returns at least one row. | Cursor\_name%FOUND |
| FALSE, if fetch statement doesn’t return a row. |
| %NOTFOUND | TRUE, if fetch statement doesn’t return a row. | Cursor\_name%NOTFOUND |
| FALSE, if fetch statement returns at least one row. |
| %ROWCOUNT | The number of rows fetched thusfar by the fetch statement | Cursor\_name%ROWCOUNT |
| If no row is returned, the PL/SQL statement returns an error. |
| %ISOPEN | TRUE, if the cursor is already open in the program | Cursor\_name%ISNAME |
| FALSE, if the cursor is not opened in the program. |

Using Loops with Explicit Cursors:

Three methods of using cursors to deal with multiple rows of data:

* SIMPLE LOOP,
* WHILE LOOP
* And FOR LOOP

These loops can be used to process multiple rows in the cursor.

Cursor with a basic loop:

*LOOP*

*Statement(s)…*

*FETCH cursor\_name INTO record\_name  
 EXIT WHEN condition*

*Statements…  
END LOOP;*

\*NOTE: condition is usually when an explicit cursor attributes is equal to some value. For example: cursor\_name%NOTFOUND

**Cursor for WHILE loop:**

*WHILE condition THEN*

*LOOP*

*Statement(s)*

*FETCH cursor\_name INTO record\_name;*

*End LOOP;*

\*NOTE: condition is usually when explicit cursor attributes are equal to some value. For example: cursor\_name%FOUND

**Cursor with a FOR Loop:**

When using FOR LOOP you DO NOT:

1. Declare a record or variables to store the cursor values,
2. OPEN, FETCH, and CLOSE the cursor

These functions are accomplished by the FOR LOOP automatically.

**Syntax:**

*--No parameters*

*FOR record\_name IN cursor\_name*

*LOOP*

*process the row...*

*END LOOP;*

*--Parameters*

*FOR record\_name IN cursor\_name(parameter values)*

*LOOP*

*Process the row…*

*END LOOP;*

## Exception Handling

When an exception occurs, a message which explains its cause is received.   
PL/SQL Exception message consists of three parts.   
1) Type of Exception  
2) An Error Code  
3) A message

By handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

### Structure of Exception Handling.

***General Syntax :***

*DECLARE*

*Declaration section*

*BEGIN*

*Execution section, that may throw exceptions*

*EXCEPTION*

*WHEN exception\_name1 THEN*

*-Error handling statements*

*WHEN exception\_name2 THEN*

*-Error handling statements*

*WHEN Others THEN*

*-Error handling statements*

*END;*

### Using PL/SQL statements in the Exception Block.

When an exception is raised:

1. Oracle searches for the exception handler in the exception section
2. If it cannot find the exception, the WHEN OTHERS exception is used.

For example, if the error raised is ‘exception\_name1’, then the error is handled according to the statements under it. Since it’s not possible to determine all the possible runtime errors during the testing of code, the ‘WHEN Others’ exception is used to manage the exceptions that are not *explicitly* handled.

Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

If the exception is raised in the inner block it should be handled in the exception area of the inner PL/SQL block, else the control moves to the Execution block of the next upper PL/SQL block. If none of the blocks handle the exception, the program ends abruptly with an error.  
  
Types of Exception.

There are 3 types of Exceptions.   
a) Named or predefined System Exceptions   
b) Unnamed or internally defined System Exceptions   
c) User-defined Exceptions

#### Named System Exceptions

Systems exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle, which are known as Named System Exceptions

Named system exceptions are:   
1) Not Declared explicitly,  
2) Raised implicitly when a predefined Oracle error occurs  
3)caught by referencing the standard name within an exception-handling routine.

|  |  |  |
| --- | --- | --- |
| **Exception Name** | **Reason** | **Error Number** |
| CURSOR\_ALREADY\_OPEN | Trying to open a cursor that is already open. | ORA-06511 |
| INVALID\_CURSOR | performing an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened. | ORA-01001 |
| NO\_DATA\_FOUND | a SELECT...INTO clause does not return any row from a table. | ORA-01403 |
| TOO\_MANY\_ROWS | When you SELECT or fetch more than one row into a record or variable. | ORA-01422 |
| ZERO\_DIVIDE | When you attempt to divide a number by zero. | ORA-01476 |

**More can be found at:**

[**http://docs.oracle.com/cd/B28359\_01/appdev.111/b28370/errors.htm**](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm)

**For Example:**Suppose a NO\_DATA\_FOUND exception is raised we can write code to handle the exception as given below.

*BEGIN*

*Execution section*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*dbms\_output.put\_line ('A SELECT...INTO did not return any row.');*

*END;*

Oracle also has two helpful functions that we can use:

1. SQLERRM – It takes an integer as its parameter and returns the error message associated with that number. It can also be called with no parameters in which case it returns the current error message text.  
   Example:  
    DBMS\_OUTPUT.PUT\_LINE(SQLERRM(-60));  
    DBMS\_OUTPUT.PUT\_LINE(SQLERRM);
2. SQLCODE – Returns the error code number for the most recently raised exception

#### Unnamed System Exceptions

There are many system exceptions, not all of which have been provided a name since they do not occur frequently. These are unnamed system exceptions. They do however have a code (or number) and an associated message.  
  
There are two ways to handle unnamed system exceptions:   
1. By using the WHEN OTHERS exception handler, or  
2. By associating the exception code to a name and using it as a named exception.

**PRAGMA** -Signifies that the statement is a compiler directive. Pragmas are processed at compile time, not at run time. They do not affect the meaning of a program; they convey information to the compiler.

To assign a name to unnamed system exceptions use a **Pragma** called **EXCEPTION\_INIT.**   
**EXCEPTION\_INIT** will associate a predefined Oracle error number to a programmer\_defined exception name.

Unnamed system exceptions:  
• are raised implicitly  
• are not handled in WHEN OTHERS must be handled explicitly  
• handled explicitly, must be declared using Pragma EXCEPTION\_INIT *as given above* and handled referencing the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION\_INIT is:

*DECLARE*

*exception\_name EXCEPTION;*

*PRAGMA*

*EXCEPTION\_INIT (exception\_name, Err\_code);*

*BEGIN*

*Execution section*

*EXCEPTION*

*WHEN exception\_name THEN*

*handle the exception*

*END;*

#### User-defined Exceptions

Apart from system exceptions, we can explicitly define exceptions based on business rules. These are known as User-Defined Exceptions.  
  
Steps to be followed to use user-defined exceptions:   
• They should be explicitly declared in the declaration section  
• They should be explicitly raised in the Execution section.  
• They should be handled by referencing the user-defined exception name in the exception section.

*DECLARE*

*Exception\_name EXCEPTION;*

*BEGIN*

*RAISE Exception\_name; --usually called as a result of a conditional statement*

*EXCEPTION*

*WHEN Exception\_name THEN*

*--do something*

*END;*

*END;*

### Raising an error

RAISE\_APPLICATION\_ERROR ( )

* A built-in procedure for displaying user-defined error messages and the error number.
* Whenever a message is displayed using RAISE\_APPLICATION\_ERROR, all previous transactions which are not committed within the PL/SQL Block are **rolled back automatically**. (Ie: change due to INSERTS, UPDATE, and DELETE statements).
* RAISE\_APPLICATION\_ERROR raises an exception but does not handle it.

RAISE\_APPLICATION\_ERROR is used for the following reasons,

* To create a unique id for a user-defined exception
* To make the user-defined exceptions look like an Oracle error.

Syntax:

RAISE\_APPLICATION\_ERROR (error\_number, error\_message);

• The Error number must be between -20000 and -20999, these will be provided by the programmer   
• The Error\_message is the message you want to display when the error occurs. Can be up to 2048 characters long.

Steps to be followed to use RAISE\_APPLICATION\_ERROR procedure:   
1. Declare a user-defined exception in the declaration section  
2. Raise the user-defined exception based on a specific business rule in the execution section  
3. Finally, catch the exception and link the exception to a user-defined error number in RAISE\_APPLICATION\_ERROR.

*DECLARE*

*Exception\_name EXCEPTION;*

*BEGIN*

*RAISE Exception\_name; --usually called as a result of a conditional statement*

*EXCEPTION*

*WHEN Exception\_name THEN*

*RAISE\_APPLICATION\_ERROR (error\_number, error\_message);*

*END;*