PL/SQL

Declarative Section

- Identified by the DECLARE keyword
- Used to define variables and constants referenced in the block
- Variable:
  - Reserve a temporary storage area in memory
  - Manipulated without accessing a physical storage medium
- Constant:Its assigned value doesn’t change during execution
- Forward execution:Variable and constants must be declared before they can be referenced

Executable Section

- Identified by the BEGIN keyword
  - Mandatory
  - Can consist of several SQL and/or PL/SQL statements
- Used to access & manipulate data within the block

Exception-handling Section

- Identified by the EXCEPTION keyword
- Used to display messages or identify other actions to be taken when an error occurs
- Addresses errors that occur during a statement's execution, Examples: No rows returned or divide by zero errors

Types of Blocks

- Procedure
- Function
- Anonymous block

Procedure

- Also called “Stored Procedures”
- Named block
- Can process several variables
- Returns no values
- Interacts with application program using IN, OUT, or INOUT parameters

Procedure Basic Syntax

```
CREATE [OR REPLACE] PROCEDURE name
[( argument [IN|OUT|IN OUT] datatype
[{, argument [IN|OUT|IN OUT] datatype }] )]
AS
/* declaration section */
BEGIN
/* executable section - required*/
EXCEPTION
/* error handling statements */
END;
/
```

Some Details

Parameter direction

- » IN (default)
- » OUT
- » IN OUT

Executing procedures

- » EXECUTE <name>(<parameters>)
- » EXECUTE <name>;

Terminal Response

```
SET SERVEROUTPUT ON;
DBMS_OUTPUT.PUT_LINE();
```

Form of argument ('Salary is: ' || salary)

Example: DBMS_OUTPUT.PUT_LINE('Name:‘|| fname);

Procedure Example

```
Create Procedure
        Insert_Customer(customerid IN varchar2)
        As
        BEGIN
     insert into customer
     values(customerid,Null, Null, Null, Null);
        END;
        /
     Execute Insert_Customer(‘c_67’);
```

Function

- Named block that is stored on the Oracle9*i* server
- Accepts zero or more input parameters
- Returns one value

Function Basic Syntax

```
CREATE [OR REPLACE] FUNCTION name
[( argument datatype
[{, argument datatype}] )]
RETURN datatype
AS
/* declaration section */
BEGIN
/* executable section - required*/
EXCEPTION
/* error handling statements */
END;
/
```

Function Example

```
CREATE OR REPLACE
FUNCTION getBDate (customerid VARCHAR2)
RETURN DATE
AS
v_bdate customer.cust_dob%TYPE;
<<customer table er cust_dob column er data type =
data type of v_bdate>>
BEGIN
SELECT cust_dob INTO v_bdate
FROM customer
WHERE cust_id = customerid;
RETURN v_bdate;
END;
/
```

Function Example

```
CREATE OR REPLACE
PROCEDURE show_date(c_id VARCHAR2)
AS
BEGIN
DBMS_OUTPUT.PUT_LINE(getBDate(c_id));
END;
/
Execute show_date('C00000000005');
```

Anonymous Block

- Not stored since it cannot be referenced by
  a name
- Usually embedded in an application
  program, stored in a script file, or
  manually entered when needed

Declaring a Variable

- Reserves a temporary storage area in the computer’s memory
- Every variable must have:
  - A name
  - A data type
  - Form is <variable> <data type>
- Variables can be initialized
- Variable name can consist of up to 30 characters, numbers, or special symbols
- Variable name must begin with a character

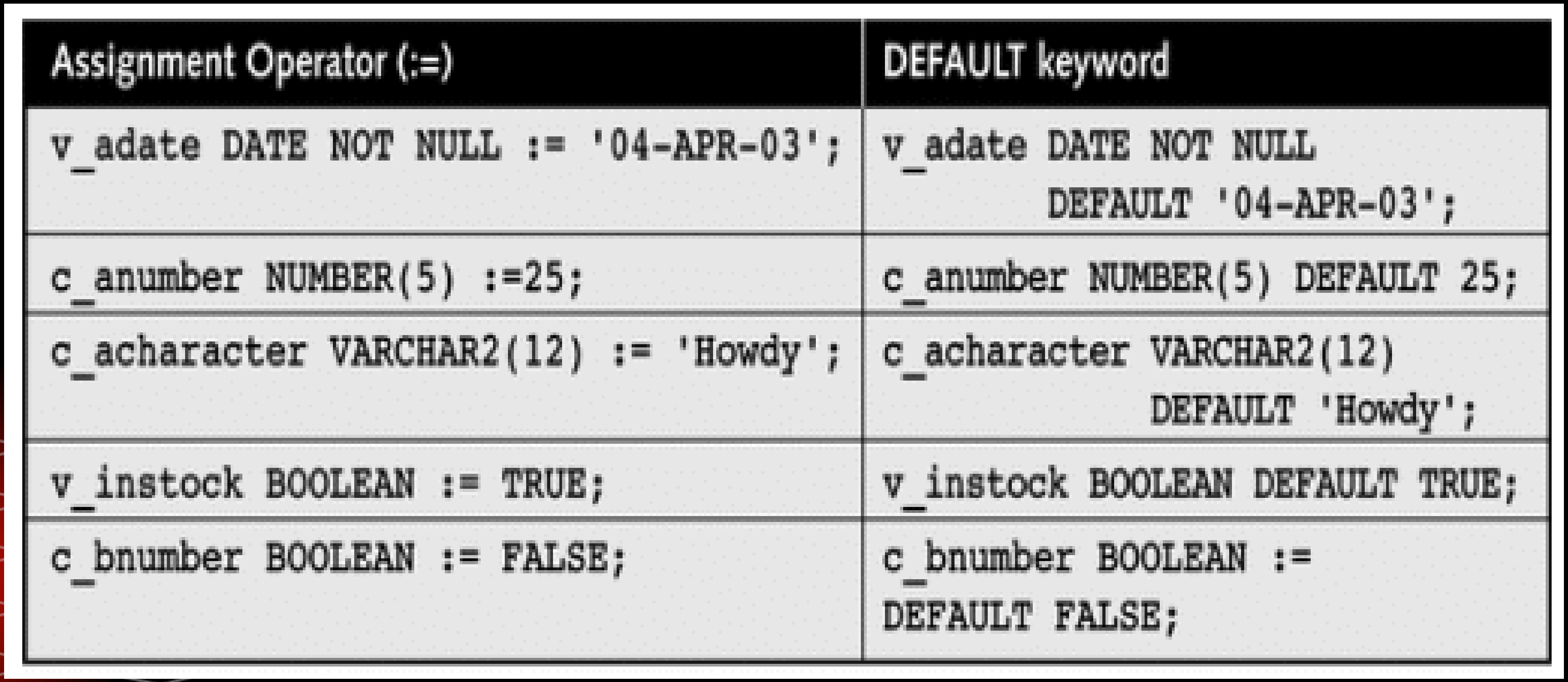
Constants

- Variables that have a value that does not change during the execution of the block
- Optional CONSTANT keyword can be used to designate a constant in the block’s declarative section

Variable Initialization

- Use DEFAULT keyword or (:=) assignment operator
- Variable must be initialized if it is assigned a NOT NULL constraint
- Non-numeric data types must be enclosed in single quotation marks

Variable Initialization Examples



Assignment Operator (:=)	DEFAULT keyword
v_adate DATE NOT NULL := '04-APR-03';	v_adate DATE NOT NULL DEFAULT '04-APR-03';
c_anumber NUMBER(5) :=25;	c_anumber NUMBER(5) DEFAULT 25;
c_acharacter VARCHAR2(12) := 'Howdy';	c_acharacter VARCHAR2(12) DEFAULT 'Howdy';
v_instock BOOLEAN := TRUE;	v_instock BOOLEAN DEFAULT TRUE;
c_bnumber BOOLEAN := FALSE;	c_bnumber BOOLEAN := DEFAULT FALSE;

%TYPE & %ROWTYPE

%TYPE

Takes the data type from the table

Form is <variable> <table>.<column>%TYPE

%ROWTYPE

Creates a record with fields for each column of

the specified table

%TYPE & %ROWTYPE

```
DECLARE
        variable name   data type;
        row_variable    table %ROWTYPE;
        BEGIN
        SELECT          column name1, column name2, …………
        INTO            row_variable
        FROM            table name
        WHERE           column name = variable name;
```

The variables are then accessed as: row_variable.column name where column name is the name of a column in the table.

SELECT Statement

The SELECT statement may be used in a block of code but the following example will return an error:

```
BEGIN
        SELECT Cust_id
        FROM Customer;
END;
/
```

SELECT Statement

Requires use of INTO clause to identify variable assigned to each data element

-

Syntax:

```
SELECT columnname [, columnname, …]
INTO variablename [, variablename, …]
FROM tablename
WHERE condition;
```

SELECT Statement Example

```
DECLARE
            S_Cust_id       VARCHAR2(12);
            S_Cust_name     VARCHAR2(12);
            S_Cust_dob      DATE;
    BEGIN
            Select Cust_id, Cust_name, Cust_dob
            Into S_Cust_id, S_Cust_name, S_Cust_dob
            From Customer
            Where Cust_id = 'C00000000005';
            DBMS_OUTPUT.PUT_LINE(‘CustomerName:’||
            S_Cust_name  || ‘Customer Date of Birth: ’ || S_Cust_dob);
    END;
    /
```

Execution Control

Decision:

IF statement

executes statements based on a condition

Loops:

Basic loop

Executes statements until condition in EXIT clause is TRUE

FOR loop

Uses counter

WHILE loop

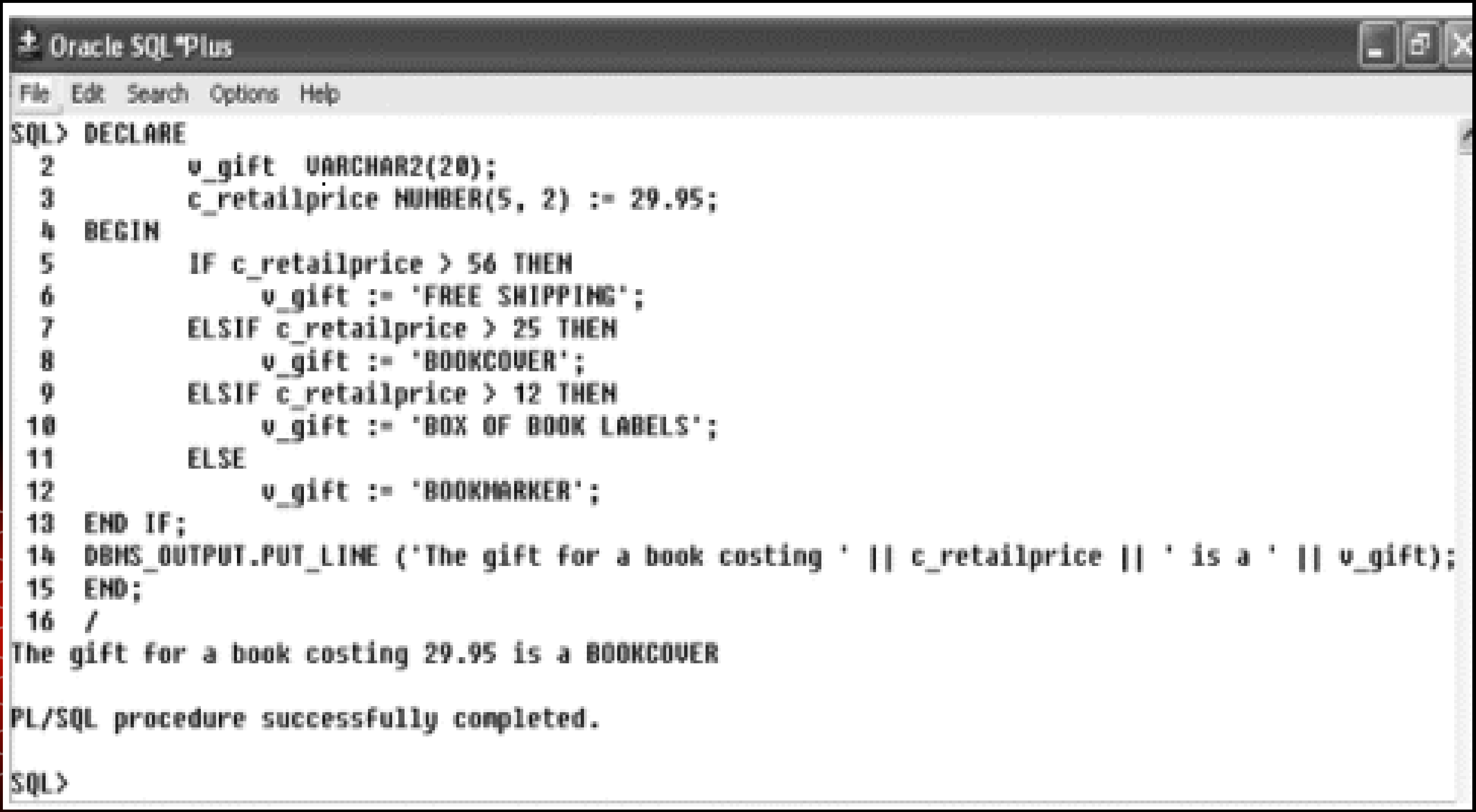
Executes statements until condition is FALSE

IF Statement

Syntax:

```
IF <condition 1> THEN
<command 1>
ELSIF <condition 2> THEN
<command 2>
ELSE
<command 3>
END IF;
```

IF Statement



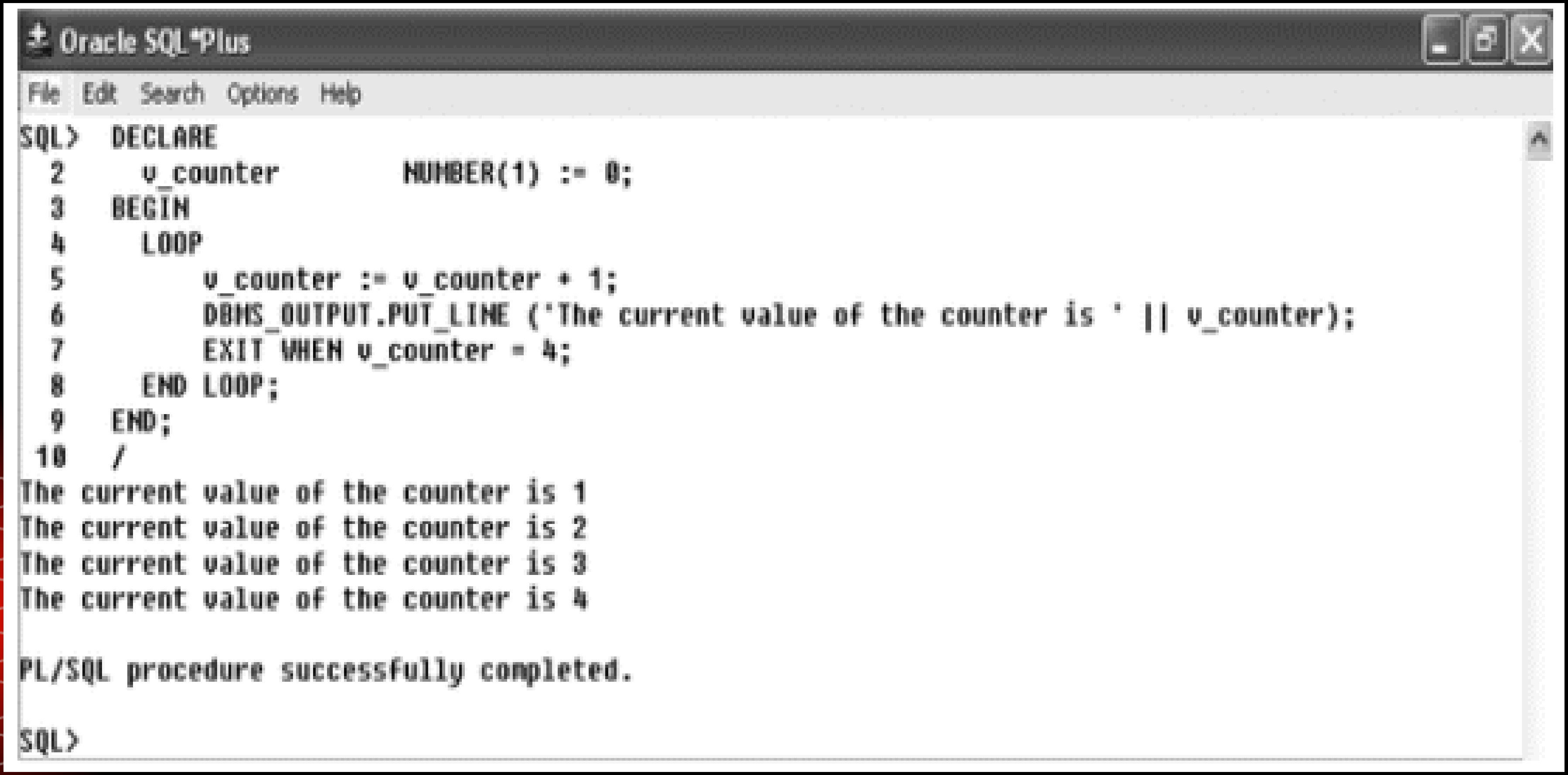
```
SQL> DECLARE
  2          v_gift  VARCHAR2(20);
  3          c_retailprice NUMBER(5, 2) := 29.95;
  4  BEGIN
  5          IF c_retailprice > 56 THEN
  6               v_gift := 'FREE SHIPPING';
  7          ELSIF c_retailprice > 25 THEN
  8               v_gift := 'BOOKCOVER';
  9          ELSIF c_retailprice > 12 THEN
 10               v_gift := 'BOX OF BOOK LABELS';
 11          ELSE
 12               v_gift := 'BOOKMARKER';
 13  END IF;
 14  DBMS_OUTPUT.PUT_LINE ('The gift for a book costing ' || c_retailprice || ' is a ' || v_gift);
 15  END;
 16  /
The gift for a book costing 29.95 is a BOOKCOVER

PL/SQL procedure successfully completed.

SQL>
```

Basic Loop

Syntax:

```
LOOP
        Statements;
        EXIT [WHEN condition];
END LOOP;
```

Basic Loop



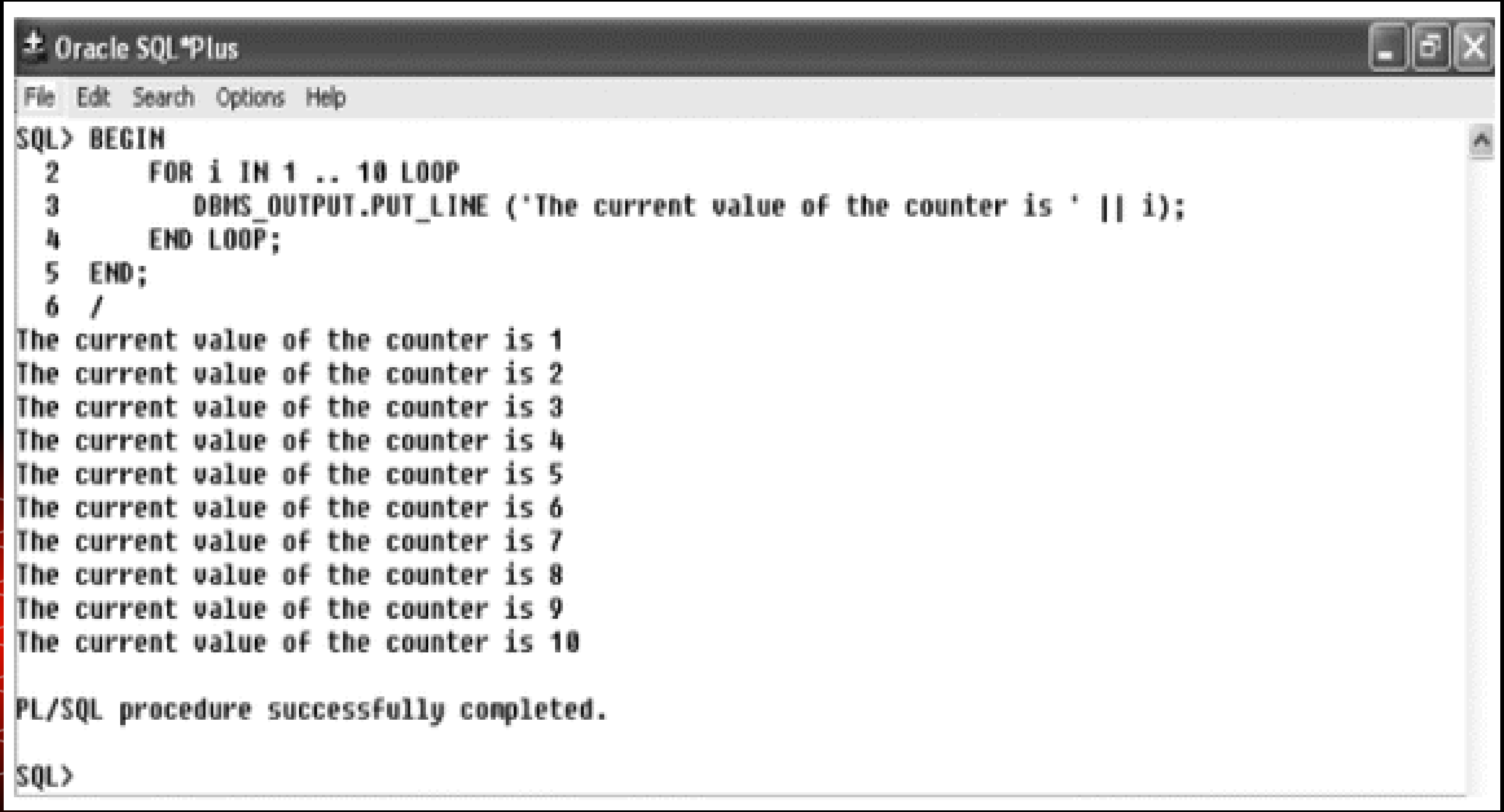
```
SQL>  DECLARE
        v_counter        NUMBER(1) := 0;
      BEGIN
        LOOP
          v_counter := v_counter + 1;
          DBMS_OUTPUT.PUT_LINE ('The current value of the counter is ' || v_counter);
          EXIT WHEN v_counter = 4;
        END LOOP;
      END;
      /
The current value of the counter is 1
The current value of the counter is 2
The current value of the counter is 3
The current value of the counter is 4

PL/SQL procedure successfully completed.

SQL>
```

FOR Loop

Syntax:

```
FOR counter IN lower_limit .. upper_limit
    LOOP
        Statements;
    END LOOP;
```

FOR Loop



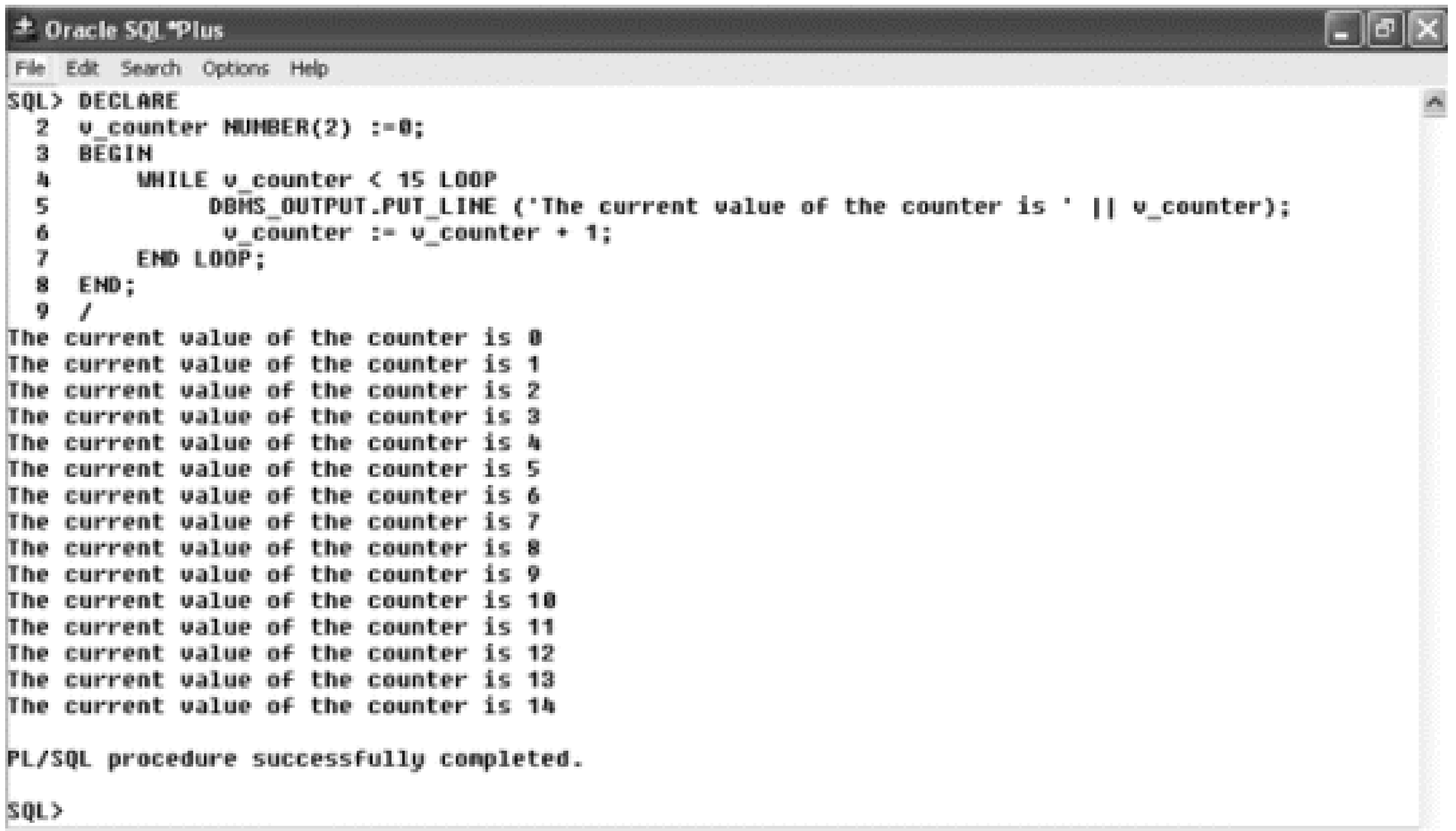
```
SQL> BEGIN
          FOR i IN 1 .. 10 LOOP
             DBMS_OUTPUT.PUT_LINE ('The current value of the counter is ' || i);
          END LOOP;
       END;
       /
The current value of the counter is 1
The current value of the counter is 2
The current value of the counter is 3
The current value of the counter is 4
The current value of the counter is 5
The current value of the counter is 6
The current value of the counter is 7
The current value of the counter is 8
The current value of the counter is 9
The current value of the counter is 10

PL/SQL procedure successfully completed.

SQL>
```

WHILE Loop

Syntax:

```
WHILE condition LOOP
        Statements;
END LOOP;
```

WHILE Loop

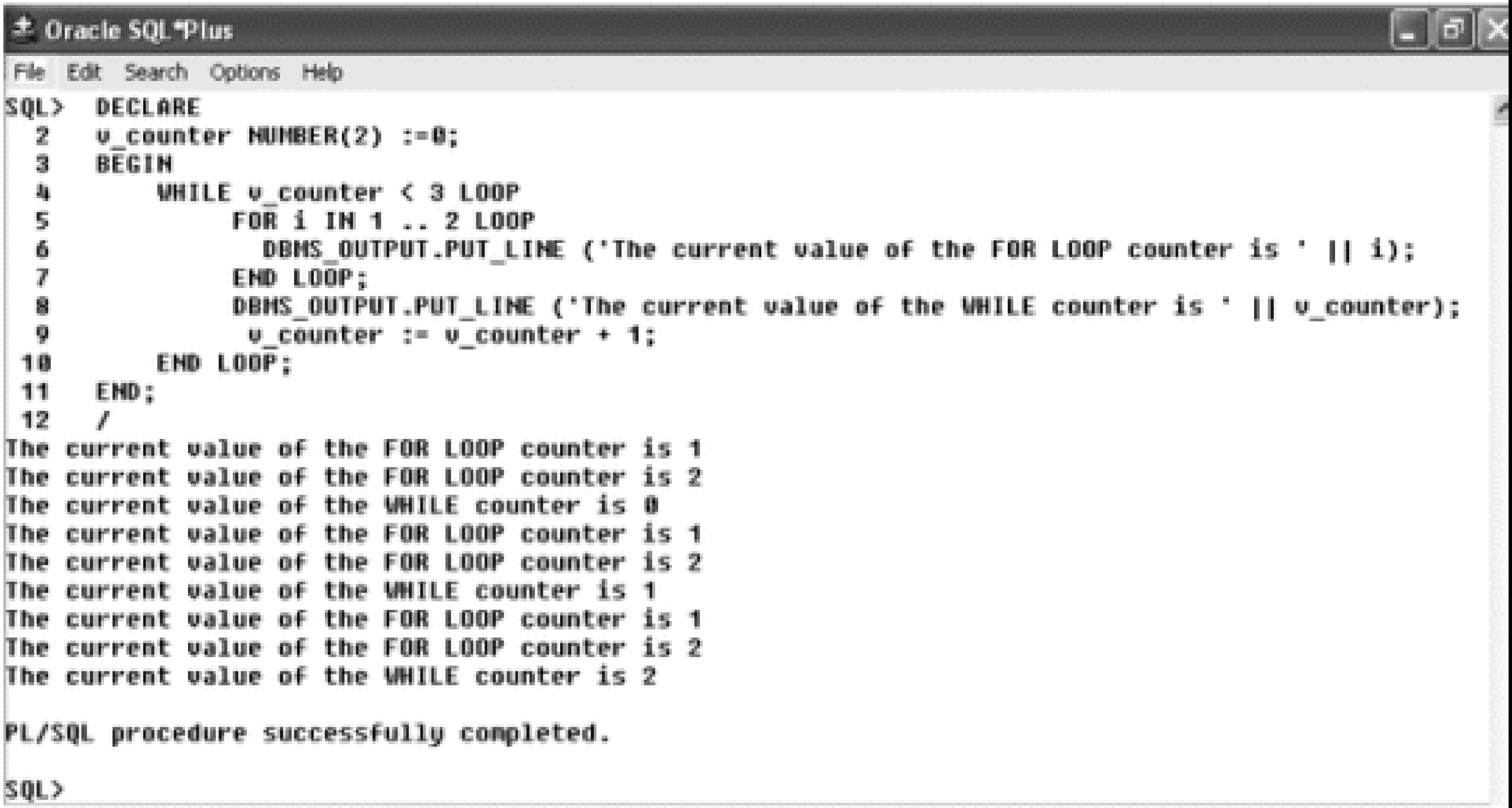


```
SQL> DECLARE
     v_counter NUMBER(2) :=0;
     BEGIN
         WHILE v_counter < 15 LOOP
               DBMS_OUTPUT.PUT_LINE ('The current value of the counter is ' || v_counter);
                v_counter := v_counter + 1;
         END LOOP;
     END;
     /
The current value of the counter is 0
The current value of the counter is 1
The current value of the counter is 2
The current value of the counter is 3
The current value of the counter is 4
The current value of the counter is 5
The current value of the counter is 6
The current value of the counter is 7
The current value of the counter is 8
The current value of the counter is 9
The current value of the counter is 10
The current value of the counter is 11
The current value of the counter is 12
The current value of the counter is 13
The current value of the counter is 14

PL/SQL procedure successfully completed.

SQL>
```

Nested Loops

- Any type of loop can be nested inside another loop
- Execution of the inner loop must be completed before control is returned to the outer loop

Nested Loops



```
SQL>  DECLARE
      v_counter NUMBER(2) :=0;
      BEGIN
          WHILE v_counter < 3 LOOP
               FOR i IN 1 .. 2 LOOP
                 DBMS_OUTPUT.PUT_LINE ('The current value of the FOR LOOP counter is ' || i);
               END LOOP;
               DBMS_OUTPUT.PUT_LINE ('The current value of the WHILE counter is ' || v_counter);
                v_counter := v_counter + 1;
          END LOOP;
      END;
      /
The current value of the FOR LOOP counter is 1
The current value of the FOR LOOP counter is 2
The current value of the WHILE counter is 0
The current value of the FOR LOOP counter is 1
The current value of the FOR LOOP counter is 2
The current value of the WHILE counter is 1
The current value of the FOR LOOP counter is 1
The current value of the FOR LOOP counter is 2
The current value of the WHILE counter is 2

PL/SQL procedure successfully completed.

SQL>
```

Exception Handling

A PL/SQL block may contain statements that specify Exception handling routines.

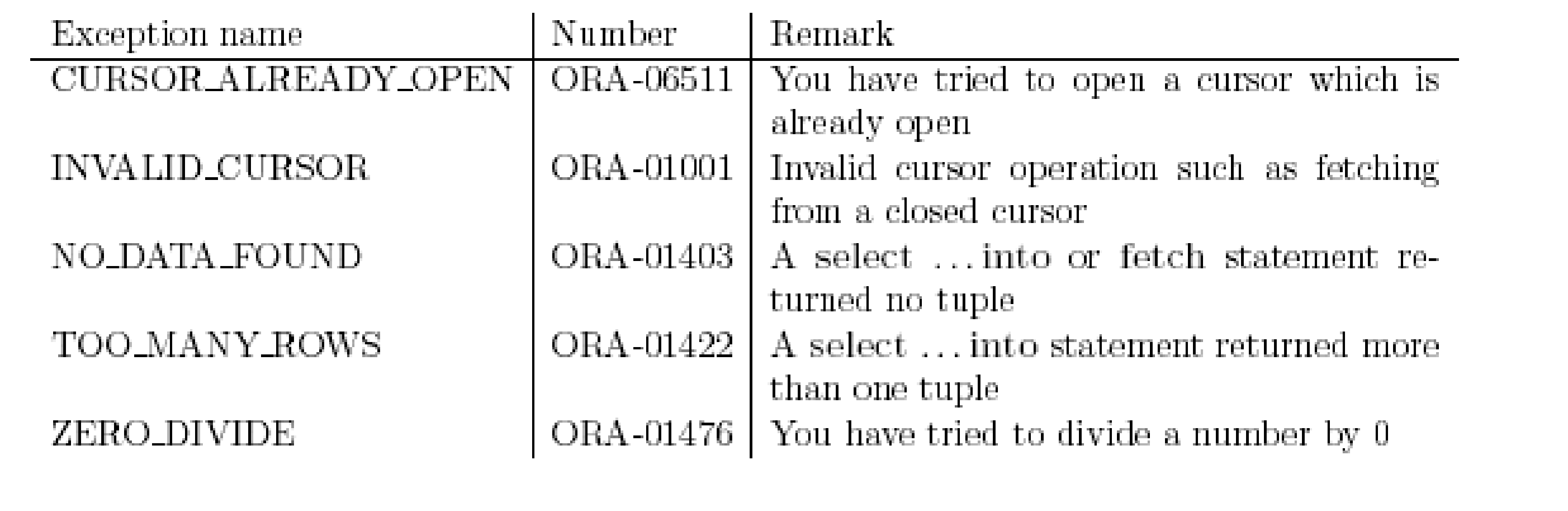
Each error or warning during the execution of a PL/SQL block raises an exception.

One can distinguish between two types of exceptions:

- system defined exceptions
- user defined exceptions

System Defined Exceptions

System defined exceptions are always automatically raised whenever corresponding errors or warnings occur.



Exception name	Number	Remark
CURSOR_ALREADY_OPEN	ORA-06511	You have tried to open a cursor which is already open
INVALID_CURSOR	ORA-01001	Invalid cursor operation such as fetching from a closed cursor
NO_DATA_FOUND	ORA-01403	A select ... into or fetch statement returned no tuple
TOO_MANY_ROWS	ORA-01422	A select ... into statement returned more than one tuple
ZERO_DIVIDE	ORA-01476	You have tried to divide a number by 0

User Defined Exceptions

User defined exceptions, in contrast, must be raised explicitly in a sequence of statements using raise <exception name>.

After the keyword exception at the end of a block, user defined exception handling routines are implemented.

An implementation has the pattern

when <exception name> then <sequence of statements>;

Exception Handling

```
declare
emp_sal      NUMBER(10,3);
Emp_no       VARCHAR2(12);
too_high_sal exception;
begin
select EMPLOYEE_ID, SALARY into emp_no, emp_sal
from EMPLOYEE where EMPLOYEE_NAME = ’E_Y’;
if emp_sal * 1.05 > 2000 then raise too_high_sal;
  end if;
exception
when NO_DATA_FOUND
then DBMS_OUTPUT.PUT_LINE(‘No data found’);
when too_high_sal then DBMS_OUTPUT.PUT_LINE(‘High Salary’);
end;
/
```

Exception Handling

It is also possible to use procedure raise_application_error.

This procedure has two parameters

<error number> and <message text>.

<error number> is a negative integer defined by the user and must range between -20000 and -20999.

<error message> is a string with a length up to 2048 characters.

If the procedure raise application error is called from a PL/SQL block, processing the PL/SQL block terminates and all database modifications are undone, that is, an implicit rollback is performed in addition to displaying the error message.

Exception Handling

```
declare
emp_sal  NUMBER(10,3);
Emp_no  VARCHAR2(12);
begin
select EMPLOYEE_ID, SALARY into emp_no, emp_sal
from EMPLOYEE where EMPLOYEE_NAME = ’E_Y’;
if emp_sal * 1.05 > 2000
then raise_application_error(-20010, ’Salary  is too high’);
end if;
end;
/
```

Compilation Errors

Loading a procedure or function may cause compilation errors.

SHOW ERRORS; gives the errors

To get rid of procedures or functions:

DROP PROCEDURE <name>;

DROP FUNCTION <name>;

Cursors

When a query returns multiple rows, defining a cursor allows us to

- » process beyond the first row returned
- » keep track of which row is currently being processed

Cursors are defined and manipulated using

- » DECLARE
- » OPEN
- » FETCH
- » CLOSE

Declaring Cursors

Syntax

- Cursor name - similar to a pointer variable
- There is no INTO clause
- Example

```
CURSOR <cursor name> IS <select-expression>;
```

```
CURSOR emp_cursor IS
SELECT employee_id, employee_name
FROM employee
WHERE employee_name LIKE 'E%';
```

Opening a Cursor

Opens a cursor (which must be closed)

Gets the query result from the database

The rows returned become the cursor's current active set

Sets the cursor to position before the first row. This becomes the current row.

NOTE - You must use the same cursor name if you want data from that cursor.

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

Fetching A Row

Syntax

Moves the cursor to the next row in the current active set

Assigns values to the host variables

```
FETCH <cursor name>
INTO <host variables>;
```

```
FETCH emp_cursor
INTO e_id, e_name;
```

Closing the Cursor

Closes the cursor (which must be open)

There is no longer an active set

Reopening the same cursor will reset it to point to the beginning of the returned table

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

CURSOR Example

```
DECLARE
    CURSOR emp_cursor IS
    SELECT employee_id, employee_name
    FROM employee
    WHERE employee_name LIKE 'E%';
    emp_val emp_cursor%ROWTYPE;
BEGIN
    OPEN emp_cursor;
    FETCH emp_cursor INTO emp_val;
    DBMS_OUTPUT.PUT_LINE(emp_val.employee_id);
    CLOSE emp_cursor;
END;
/
```

Cursor Properties

Cursors have four attributes that can be used

In program:

- –%FOUND, %NOTFOUND : a record can/cannot be fetched from the cursor
- –%ISOPEN : the cursor has been opened
- –%ROWCOUNT : the number of rows fetched from the cursor so far

Simple Cursor Loops

```
DECLARE
    CURSOR emp_cursor IS
    SELECT employee_id, employee_name FROM employee
    WHERE employee_name LIKE 'E%';
    emp_val emp_cursor%ROWTYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
            FETCH emp_cursor INTO emp_val;
            EXIT WHEN emp_cursor%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(emp_val.employee_id);
    END LOOP;
    CLOSE emp_cursor;
END;
/
```

Cursor FOR Loops

```
DECLARE
    CURSOR emp_cursor IS
    SELECT employee_id, employee_name
    FROM employee
    WHERE employee_name LIKE 'E%';
BEGIN
    for emp_val in emp_cursor
    LOOP
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(emp_val.employee_id);
    END LOOP;
END;
/
```

Cursor FOR Loops

- In a Cursor FOR Loop, there is no open or fetch command.
- The command for emp_val in emp_cursor implicitly opens the emp_cursor cursor and fetches a value into the emp_val variable.
- When no more records are in the cursor, the loop is exited and the cursor is closed.
- In a Cursor FOR loop, there is no need for a close command.
- Note that emp_val is not explicitly declared in the block.

GOTO Statement

The GOTO statement transfers control to a labeled block or statement. If a GOTO statement exits a cursor FOR LOOP statement prematurely, the cursor closes.

Restrictions on GOTO Statement:

1. A GOTO statement cannot transfer control into an IF statement, CASE statement, LOOP statement, or sub-block.
2. A GOTO statement cannot transfer control from one IF statement clause to another, or from one CASE statement WHEN clause to another.
3. A GOTO statement cannot transfer control out of a subprogram.
4. A GOTO statement cannot transfer control into an exception handler.
5. A GOTO statement cannot transfer control from an exception handler back into the current block (but it can transfer control from an exception handler into an enclosing block).

```
DECLARE
  p  VARCHAR2(30);
  n  PLS_INTEGER := 15;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN
      p := ' is not a prime number';
      GOTO print_now;
    END IF;
  END LOOP;

  p := ' is a prime number';

  <<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

```
15 is not a prime number
```

Example : GOTO Statement

EXIT Statement

- The EXIT statement exits a loop and transfers control to the end of the loop.
- The EXIT statement has two forms: the unconditional EXIT and the conditional EXIT WHEN. With either form, you can name the loop to be exited.
- If and only if the value of this expression is TRUE, the current loop (or the loop labeled by label_name) is exited immediately.

```
SQL> DECLARE
2 x NUMBER := 0;
BEGIN
LOOP
     DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' || TO_CHAR(x));
x := x + 1;
IF x > 3 THEN
EXIT;
END IF;
END LOOP;
-- After EXIT, control resumes here
DBMS_OUTPUT.PUT_LINE (' After loop: x = ' || TO_CHAR(x));
END;
/
Inside loop: x = 0
Inside loop: x = 1
Inside loop: x = 2
Inside loop: x = 3
After loop: x = 4
```

Example: EXIT Statement

PL/SQL - Cursors

- A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.
- You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:
- Implicit cursors
- Explicit cursors

Implicit Cursors

- In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**.
- The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement.

```
DECLARE
total_rows number(2);
BEGIN
UPDATE customers
SET salary = salary + 500;
IF sql%notfound THEN
dbms_output.put_line('no customers selected'); ELSIF sql%found THEN
total_rows := sql%rowcount;
dbms_output.put_line( total_rows || ' customers selected ');
END IF;
END;
/
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected.

Example: Implicit Cursors

Explicit Cursors

- An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.
- Declaring the cursor defines the cursor with a name and the associated SELECT statement.

```
DECLARE
c_id customers.id%type;
c_name customerS.No.ame%type;
c_addr customers.address%type;
CURSOR c_customers is SELECT id, name, address FROM
   customers;
BEGIN OPEN c_customers;
LOOP FETCH c_customers into c_id, c_name, c_addr;
EXIT WHEN c_customers%notfound;
   dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

Example: Explicit Cursors

Parameterized Cursors

- Parameterized cursors are static cursors that can accept passed-in parameter values when they are opened.

```
DECLARE
my_record emp%ROWTYPE;
CURSOR c1 (max_wage NUMBER)
IS SELECT * FROM emp WHERE sal < max_wage;
BEGIN OPEN c1(2000);
LOOP FETCH c1 INTO my_record;
EXIT WHEN c1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
|| my_record.sal);
END LOOP;
CLOSE c1;
END;
```

PLSQL: Cursor Attributes

- While dealing with cursors, you may need to determine the status of your cursor. The following is a list of the cursor attributes that you can use.
- %ISOPEN

  Returns TRUE if the cursor is open, FALSE if the cursor is closed.
- %FOUND
- Returns INVALID_CURSOR if cursor is declared, but not open; or if cursor has been closed.
- Returns NULL if cursor is open, but fetch has not been executed.

  Returns TRUE if a successful fetch has been executed.

  Returns FALSE if no row was returned.

- %NOTFOUND

  - Returns INVALID_CURSOR if cursor is declared, but not open; or if cursor has been closed.

  - Return NULL if cursor is open, but fetch has not been executed.

  - Returns FALSE if a successful fetch has been executed.

  - Returns TRUE if no row was returned.

- %ROWCOUNT

  - Returns INVALID_CURSOR if cursor is declared, but not open; or if cursor has been closed.

  - Returns the number of rows fetched.

  - The ROWCOUNT attribute doesn't give the real row count until you have iterated through the entire cursor. In other words, you shouldn't rely on this attribute to tell you how many rows are in a cursor after it is opened.

```
CREATE OR REPLACE Function FindCourse
( name_in IN varchar2 )
RETURN number IS cnumber number;
CURSOR c1
IS SELECT course_number
FROM courses_tbl
WHERE course_name = name_in;
BEGIN open c1;
fetch c1 into cnumber;
if c1%notfound then cnumber := 9999;
end if;
close c1;
RETURN cnumber;
END;
```

Example: How to use the %NOTFOUND attribute.