PL/SQL Variables and Data Types

- Variable names must begin with a letter
- maximum 30 characters
- Variables can contain letters, numbers, and symbols $, _, and #.
- cannot be reserved words such as NUMBER, VALUES, BEGIN, etc.
- variable names cannot be the same as database table names

Scalar Variables

Scalar variables reference a single value, such as a number, date, or character string.

Data Types	Usage	Sample Declaration
Varchar2	Variable-length character strings	StudentName VARCHAR2(30)
CHAR	Fixed-length character strings	StudentGender CHAR(1)
NUMBER	Floating, fixed-point, or integer numbers	CurrentPrice NUMBER(5,2);
DATE	Dates	TodaysDate DATE;
LONG	Test, up to 32,760 bytes	Evaluation_summary LONG;

Other scalar data types that do not correspond to database data types

Data Types	Usage	Sample Declaration
BINARY_INTEGER, INTEGER, INT, SMALLINT	Integers	CustID BINARY_INTEGER;
DEC, DECIMAL, DOUBLE PRECISION, NUMERIC, REAL	Numeric values with varying precision and scale	Student_gpa REAL;
BOOLEAN	True/False values	OrderFlag BOOLEAN;

Composite Variables

A composite variable references a data structure that contains multiple scalar variables, such as a record or a table.

- **RECORD** – contains multiple scalar values
- **TABLE** – tabular structure with multiple rows and columns
- **VARRAY** – variable-sized array, which is a tabular structure that can expand and contract based on the data values it contains.

Reference Variables

Reference variables directly reference a specific database field or record and assume the data type of the associated field or record.

Data Types	Usage	Sample Declaration
%Type	Assumes the data type of a database field	CustAddress customer.cadd%TYPE
%ROWTYPE	Assumes the data type of a database row	CustOrderRecord cust_order%ROWTYPE;

- General format for a %TYPE data declaration is :

  <variable name> <tabel name>.<field name>%TYPE

  e.g. LNAME FACULTY.FLNAME%TYPE

- General format of for %ROWTYPE data declaration is:

  <row variable name> <table name>%ROWTYPE;

  e.g. FacRow FACULTY%ROWTYPE;

LOB Data Types

Used to declare variables that reference binary data objects, such as images or sounds.

Large Object (LOB) – binary or character data up to 4GB

PL/SQL Program Blocks

```
DECLARE
    <variable declarations>
BEGIN
    <executable commands>
EXCEPTION
    <error-handling statements>
END;
```

```
    DECLARE
            pi        CONSTANT NUMBER(9,7) := 3.1415926;
            radius    INTEGER(5);
            area      NUMBER(14,2);
    BEGIN
            radius := 3;
            area := pi*power(radius,2);
            insert into AREAS values (radius, area);
    END;
    .
    /
```

```
select * from AREAS;

    RADIUS        AREA
    ------------    ----------------
                3               28.27
```

Comments in PL/SQL

- A block of comments that spans several lines can begin with the symbols /* and end with the symbols */.

```
/* Script:   orcl_cold_backup
   purpose           To perform a complete cold backup on the ORCL database
instance  */
```

- If a comment statement appears on a single line, you can delimit it by typing two hyphens at the beginning of the line:

```
DECLARE
-- variable to hold current value of SID
StudentID      NUMBER;
```

PL/SQL Arithmetic Operators

**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

PL/SQL Assignment Statements

```
Sname := 'John';                /* Variable assigned to literal value */

Sname := CurrentStudentName     /* Variable assigned to another variable */
```

PL/SQL Interactive output

- DBMS_OUTPUT package can be used for PL/SQL output.
- To use DBMS_OUTPUT, you must issue the command

```
Set Serveroutput   ON SIZE buffer_size
```

- DBMS_OUTPUT allows you to use three debugging functions within your package:

PUT	puts multiple outputs on the same line
PUT_LINE	Puts each output on a separate line
NEW_LINE	used with PUT; signals the end of the current output line

```
DBMS_OUTPUT.PUT_LINE(<text to be displayed>);
```

Writing a PL/SQL Program

```
Set serveroutput ON SIZE 4000

-- PL/SQL program to display the current date
DECLARE
     TodaysDate              DATE;
BEGIN
        TodaysDate :=  sysdate;
        DBMS_OUTPUT.PUT_LINE('Today''s date is ');
        DBMS_OUTPUT.PUT_LINE(TodaysDate);
END;
```

Data Type Conversion Functions

TO_DATE	converts a character string to a date
TO_NUMBER	converts a character string to a number
TO_CHAR	Converts either a number or a date to a character string

TO_CHAR(date[,'format'[,'NLSparameters']])

TO_CHAR(sysdate, 'MM/DD/YYYY');

TO_DATE(string[,'format'[,'NLSparameters]])

TO_DATE('22-FEB-01', 'DD-MON-YY')

Concatenating Character Strings

Variable Names	Values
SfirstName	Sarah
SLastName	Miller

SFullName := SFirstName || SLastName;

Variable Name	Data Type	Value
BuildingCode	VARCHAR2	CR
RoomNum	VARCHAR2	101
RoomCapacity	NUMBER	150

Display a message: "CR Room 101 has 150 seats"

RoomMessage := BuildingCode || ' Romm ' || RoomNum || ' has ' ||
TO_CHAR(RoomCapacity) || ' Seats.';

Set serveroutput on

```
-- PL/SQL program to display the current date
DECLARE
     TodaysDate                DATE;
BEGIN
          TodaysDate :=  sysdate;
          DBMS_OUTPUT.PUT_LINE('Today''s date is ' || TO_CHAR(Todaysdate));
END;
.
/
```

Placing String Output on a New Line

```
RoomMessage := BuildingCode || ‘ Room ‘ || RoomNum || ‘ has ‘ ||
                        TO_CHAR(RoomCapacity) || ‘ seats.’ ||
                     CHR(13) || CHR(10);
```

The IF/THEN Structure

```
IF <some condition>
  THEN  <some command>
ELSIF <some condition>
  THEN <some command>
ELSE  <some command>
END IF;
```

Nested IF

```
IF  <some condition>
  THEN
       IF   <some condition>
          THEN  <some command>
       END IF;
  ELSE  <some command>
END IF;
```

```
--PL/SQL program to display the current day
DECLARE
  Today          VARCHAR2(9);
BEGIN
  Today := TO_CHAR(SYSDATE, 'DAY');
  Today := RTRIM(Today);
 --add IF/THEN statement to determine if current day is Friday
 IF Today = 'FRIDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Friday');
 END IF;
END;
/
```

Fig. 4-22

```
--PL/SQL program to display the current day
DECLARE
   Today          VARCHAR2(9);
BEGIN
   Today := TO_CHAR(SYSDATE, 'DAY');
   Today := RTRIM(Today);
  --add IF/THEN statement to determine if current day is Friday
  IF Today != 'FRIDAY' THEN
    DBMS_OUTPUT.PUT_LINE('Today is not Friday');
  END IF;
END;
/
```

Fig. 4-23

```
--PL/SQL program to display the current day
DECLARE
   Today          VARCHAR2(9);
BEGIN
   Today := TO_CHAR(SYSDATE, 'DAY');
   Today := RTRIM(Today);
  --add IF/THEN statement to determine if current day is Friday
  IF Today = 'FRIDAY' THEN
    DBMS_OUTPUT.PUT_LINE('Today is Friday');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Today is not Friday');
  END IF;
END;
/
```

Fig. 4-24

```
DECLARE
  Today          VARCHAR2(9);
BEGIN
  Today := TO_CHAR(SYSDATE, 'DAY');
  Today := RTRIM(Today);
 --add IF/THEN statement to determine if current day is Friday
 IF Today = 'FRIDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Friday');
 ELSE
   IF Today = 'SATURDAY' THEN
     DBMS_OUTPUT.PUT_LINE('Today is Saturday');
   ELSE
     DBMS_OUTPUT.PUT_LINE('Today is not Friday or Saturday');
   END IF;
 END IF;
END;
/
```

Fig. 4-25

```
--PL/SQL program to display the current date  Fig. 4-26
DECLARE
  Today          VARCHAR2(9);
BEGIN
  Today := TO_CHAR(SYSDATE, 'DAY');
  Today := RTRIM(Today);
 --add IF/ELSIF statement to display current day
 IF Today = 'FRIDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Friday');
 ELSIF Today = 'SATURDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Saturday');
 ELSIF Today = 'SUNDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Sunday');
 ELSIF Today = 'MONDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Monday');
 ELSIF Today = 'TUESDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Tuesday');
 ELSIF Today = 'WEDNESDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Wednesday');
 ELSIF Today = 'THURSDAY' THEN
   DBMS_OUTPUT.PUT_LINE('Today is Thursday');
 ELSE
   DBMS_OUTPUT.PUT_LINE('Error displaying day of week');
 END IF;
END;
/
```

Data Definition Language (DDL) : Changes the database structure – CREATE, ALTER, DROP, GRANT, REVOKE

Data Manipulation Language (DML): Queries or changes the data in the database table – SELECT, INSERT, UPDATE, DELETE.

Transaction Control commands: Organise commands into logical transactions- COMMIT, ROLLBACK, SAVEPOINT

```
Start e:\data\ch4\emptynorthwoods.sql
set serveroutput on

DECLARE
     Term_id          Binary_Integer     := 1;
     Term_desc        VARCHAR2(20)  := 'Fall 2002';
     Term-status      VARCHAR2(20)  := 'CLOSED';

BEGIN
    -- insert the records
    INSERT INTO term VALUES( term_id, term_desc, term_status);
    -- update values and  insert other records
    Term_id := term_id+1;
    Term_desc := 'Spring 2003';
    INSERT INTO term VALUES(term_id, term_desc, term_status);
    Term_id := term_id+1;
    Term_desc := 'Summer 2003';
    INSERT INTO term VALUES(term_id, term_desc, term_status);
    COMMIT;
END;
/


select * from term;

TERM_ID     TERM_DESC          STATUS
----------          ----------          ------------
         1        Fall 2002             CLOSED
         2        Spring 2003           CLOSED
         3        Summer 2003           CLOSED
```

Simple loops: A loop that keeps repeating until an **exit** or **exit when** statement is reached within the loop

FOR loops: A loop that repeats a specified number of times

WHILE loops: A loop that repeats until a condition is met

Pretest Loop

```
CREATE TABLE  count_table (
      counter      NUMBER(2)
);
```

```
DECLARE
        Loopcount      BINARY_INTEGER;
BEGIN
        Loopcount := 0;
        LOOP
                Loopcount := Loopcount + 1;
                IF Loopcount = 6 THEN
                        EXIT;
                END IF;
                Insert into count_table values (Loopcount);
        END LOOP;
END;
/

SELECT * FROM count_table;

COUNTER
---------
       1
       2
       3
       4
       5
```

Posttest Loop

```
Create table AREAS (
      Radius     NUMBER(5),
      AREAS    NUMBER(6,2)
);

DECLARE
          pi        constant  NUMBER(9,7) := 3.1415926;
          radius  INTEGER(5);
          area     NUMBER(14,2);
BEGIN
          radius := 3;
          LOOP
              area := pi*power(radius,2);
                    insert into AREAS values(radius, area);
                radius := radius+1;
                EXIT WHEN area >100;
          END LOOP;
END;
/

select * from AREAS
    order by Radius;
```

RADIUS	AREA
3	28.27
4	50.27
5	78.54
6	113.1

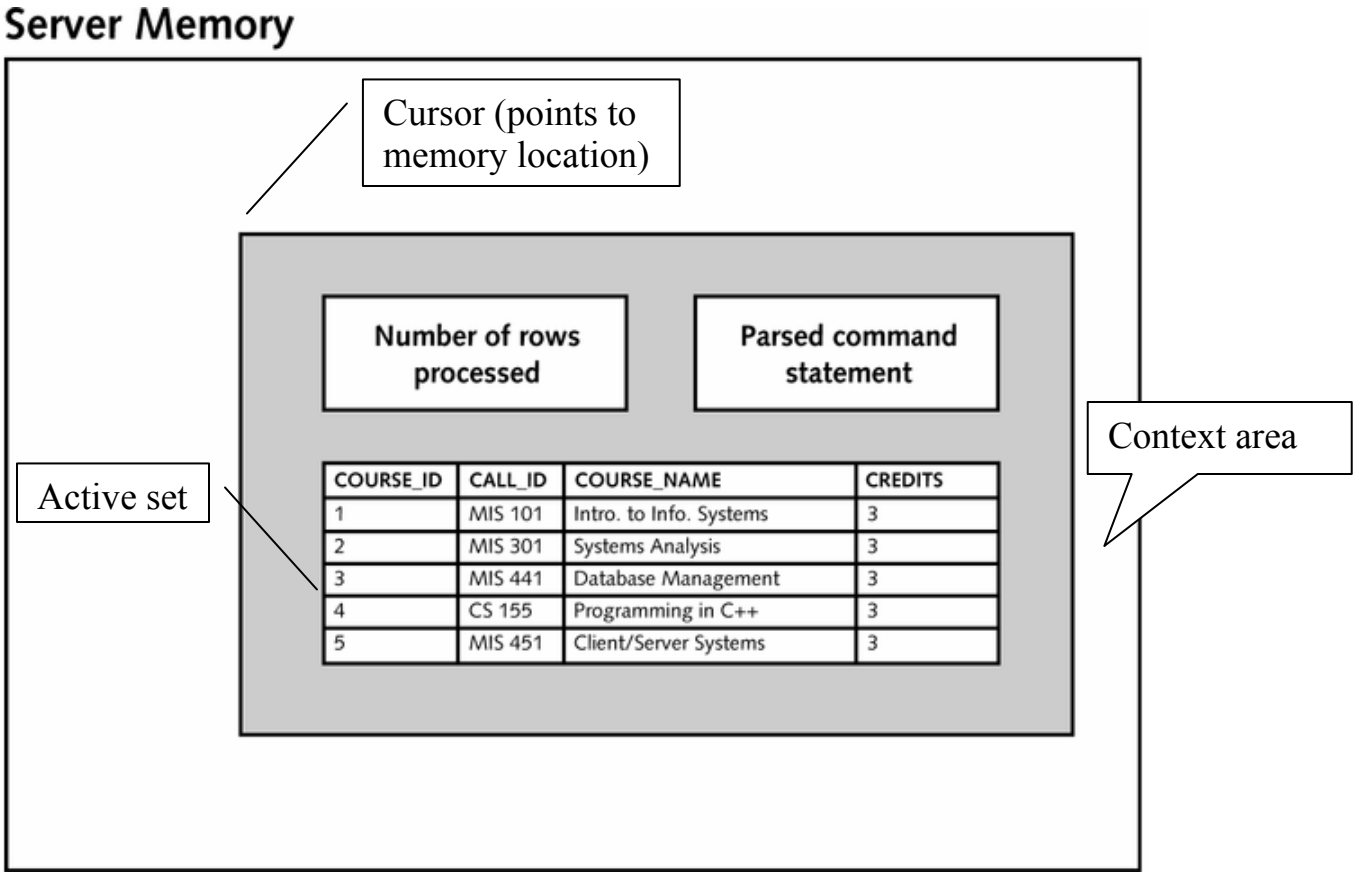
WHILE Loops

```
DECLARE
        Loop_count     BINARY_INTEGER  := 1;
BEGIN
        WHILE  loop_count < 6
        LOOP
                INSERT INTO count_table VALUES(loop_count);
                Loop_count := loop_count + 1;
        END LOOP;
END;
/


SELECT * FROM  count_table;

COUNTER
-------------
      1
      2
      3
      4
      5
```

For Loops

```
FOR <counter variable> IN <start value> .. <end value>
LOOP
       <program statements>
END LOOP;


DECLARE
        pi        constant  NUMBER(9,7) := 3.1415926;
        radius  INTEGER(5);
        area     NUMBER(14,2);
BEGIN
      FOR radius IN 1..7
       LOOP
          area := pi*power(radius,2);
              insert into AREAS values (radius,area);
       END LOOP;
END;
/
```

Cursors:



Implicit cursors

```
DECLARE
  Current_f_last        faculty.f_last%TYPE;
  Current_f_first       faculty.f_first%TYPE;
BEGIN
   select  f_last, f_first
   into  Current_f_last , Current_f_first
   from   faculty
   where f_id = 1;

   DBMS_OUTPUT.PUT_LINE('The faculty member''s name is ' || Current_f_first || ' '
      || Current_f_last  );
END;
/
```

The faculty member's name is Kim Cox

PL/SQL procedure successfully completed.

```
DECLARE
Current_f_last          faculty.f_last%TYPE;
  Current_f_first       faculty.f_first%TYPE;
BEGIN
   select  f_last, f_first
   into  Current_f_last , Current_f_first
   from   faculty;

   DBMS_OUTPUT.PUT_LINE('The faculty member''s name is ' || Current_f_first || ' '
      || Current_f_last  );
END;
/

DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 5
```

Fig. 4.35: Implicit cursor that returns multiple records

```
DECLARE
  Current_f_last        faculty.f_last%TYPE;
  Current_f_first       faculty.f_first%TYPE;
BEGIN
   select  f_last, f_first
   into  Current_f_last , Current_f_first
   from   faculty
   where f_id = 6;

   DBMS_OUTPUT.PUT_LINE('The faculty member''s name is ' || Current_f_first || ' '
      || Current_f_last  );
END;
/

DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5
```

Fig. 4.36: Implicit cursor that returns no records

Explicit cursors

The steps for using an explicit cursor are:

1. Declare the cursor
2. Open the cursor
3. Fetch the cursor results into PL/SQL program variables.
4. Close the cursor

```
select * from radius_vals;

RADIUS
---------
      3

DECLARE
        pi       constant  NUMBER(9,7) := 3.1415926;
        area    NUMBER(14,2);

        CURSOR rad_cursor IS
              select * from RADIUS_VALS;

          rad_val    rad_cursor%ROWTYPE;
BEGIN
        open     rad_cursor;
        fetch     rad_cursor  into rad_val;
             area := pi*power(rad_val.radius,2);
               insert into AREAS values (rad_val.radius, area);
          close rad_cursor;
END;
/
```

The query search condition can contain PL/SQL variables that have been assigned values, as long as the variables are declared before the cursor is declared.

```
DECLARE
        CurrentBidgCode              varchar2(5);

        CURSOR     locationcursor  IS
            Select locid, room, capacity
            from  location
            where bldg-code = CurrentBldgCode;
```

In the variable declaration:

rad_val rad_cursor%ROWTYPE;

the "rad_val" variable will be able to reference each column of the query's result set.

If you use %TYPE declaration, then the variable only inherits the definition of the column used to define it. You can even base %TYPE definitions on cursors, as :

```
cursor  rad_cursor is
   select  * from RADIUS_VALS;

rad_val    rad_cursor%ROWTYPE;
rad_val_radius    rad_val.Radius%TYPE;
```

The "rad_val_radius" variable inherits the data type of the Radius column within the "rad_val" variable.

Cursor's attributes

%FOUND	A record can be fetched from the cursor
%NOTFOUND	No more records can 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
        pi       constant  NUMBER(9,7) := 3.1415926;
        area     NUMBER(14,2);

        CURSOR rad_cursor IS
              select * from RADIUS_VALS;
          rad_val    rad_cursor%ROWTYPE;
BEGIN
        open     rad_cursor;
        LOOP
        fetch     rad_cursor  into rad_val;
             exit when rad_cursor%NOTFOUND;
             area := pi*power(rad_val.radius,2);
                insert into AREAS values (rad_val.radius, area);
        END LOOP;
        close rad_cursor;
END;
/
```

```
select * from RADIUS_VALS
        order by Radius;

RADIUS
-------------
          3
          4
          10

SELECT * FROM areas
      ORDER BY Radius;

RADIUS        AREA
-------------        ----------
          3         28.27
          4         50.27
          10        314.16
```

Cursor FOR Loops

```
FOR cursor_variable(s) IN cursor_name
LOOP
        Processing statements
END LOOP;


DECLARE
        pi        constant  NUMBER(9,7) := 3.1415926;
        area      NUMBER(14,2);

        CURSOR rad_cursor IS
              select * from RADIUS_VALS;
          rad_val    rad_cursor%ROWTYPE;
BEGIN
        FOR rad_val IN rad_cursor
          LOOP
             area := pi*power(rad_val.radius,2);
              insert into AREAS values (rad_val.radius, area);
        END LOOP;
END;
.
/
```

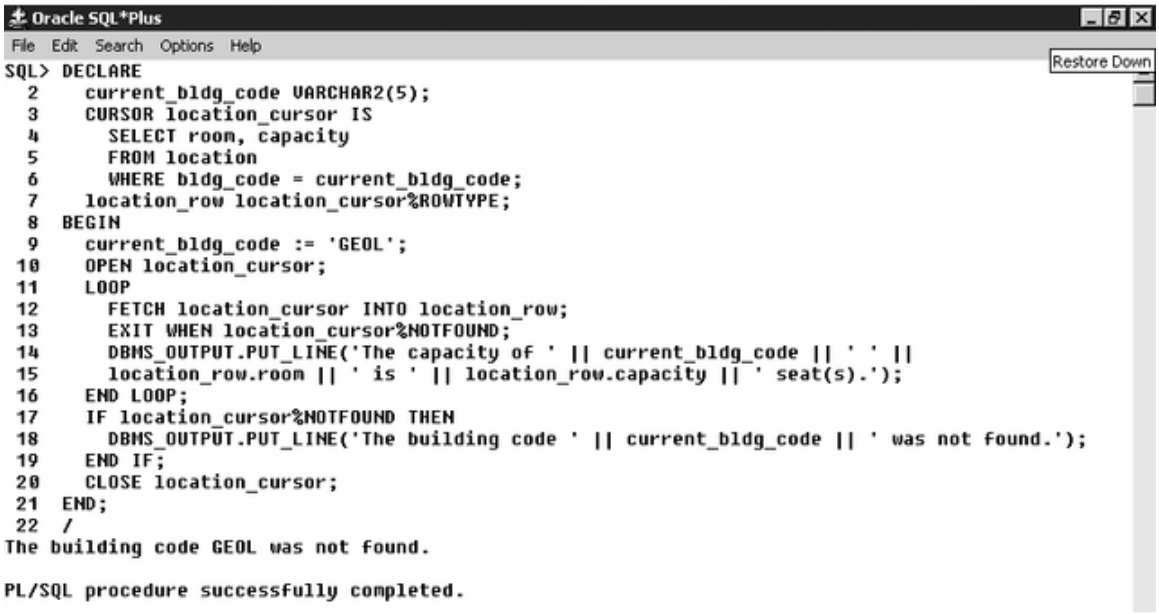


Figure 4-40: Using the %NOTFOUND attribute to generate a custom error message

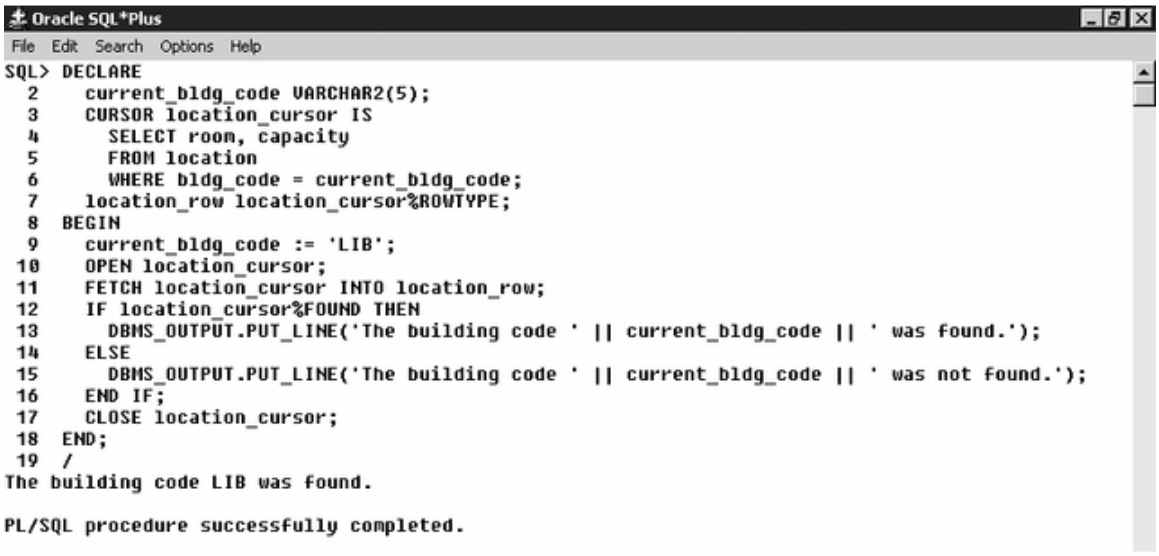


Figure 4-41: Using the %FOUND attribute to signal whether or not a cursor returns records

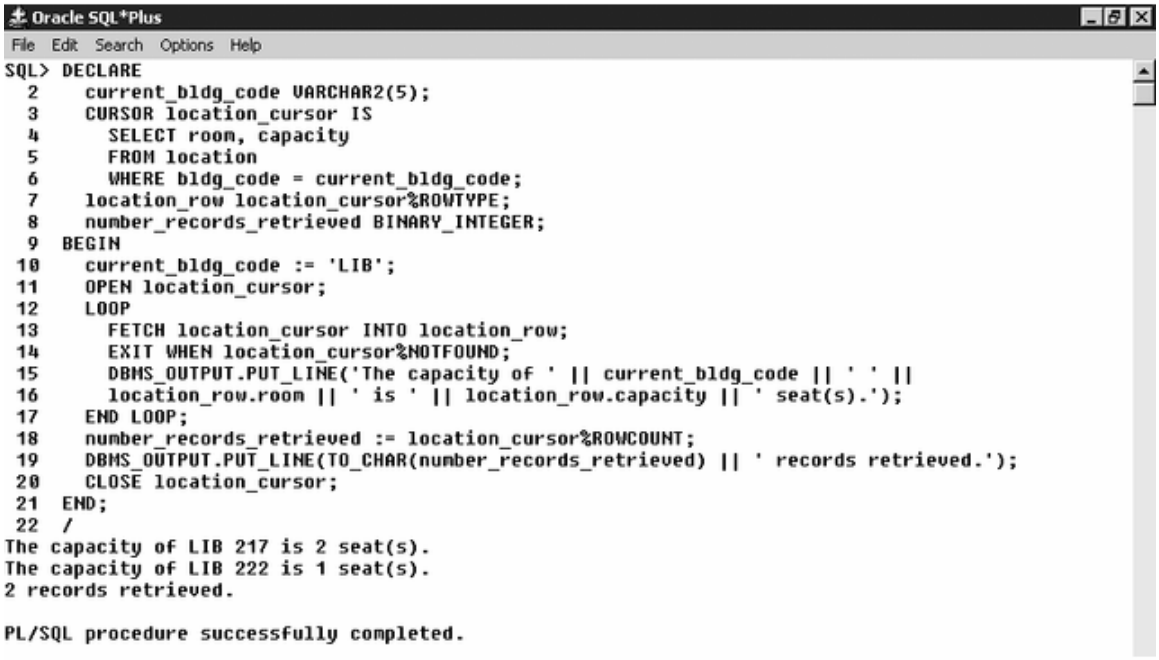


Figure 4-42: Using the %ROWCOUNT attribute to display the number of records fetched

SELECT FOR UPDATE Cursors

SELECT cursor_fieldnames
FROM tablenames
WHERE search_condition(s)
AND join_conditions
FOR UPDATE; | FOR UPDATE OF field_names;

Processing instructions

COMMIT; | ROLLBACK;

To perform the update operation during cursor processing, use the following WHERE clause

WHERE CURRENT OF cursor_name

to reference the current row being processed by the cursor.

```
DECLARE
        Current_bldg_code     VARCHAR2(5);

        CURSOR location_cursor IS
                SELECT room, capacity
                FROM location
                WHERE bldg_code = current_bldg_code
                FOR UPDATE OF capacity;

        Location_row            location_cursor%ROWTYPE;
BEGIN
        Current_bldg_code := 'LIB';

        FOR location_row IN location_cursor
        LOOP
                UPDATE location
                SET capacity = capacity + 1
                WHERE CURRENT OF location_cursor;
        END LOOP;
        COMMIT;
END;
/
```

```
SELECT * FROM location
WHERE bldg_code =  'LIB';
```

LOC_ID	BLDG_CODE	ROOM	CAPACITY
56	LIB	217	3
57	LIB	222	2

PL/SQL Tables

PL/SQL Table – data structure containing multiple data items having the same data type.

Comprised of two elements – **key**, which is a BINARY_INTEGER is unique, and the **value**, which is the actual data value.

Key	Value
1	Shadow
2	Dusty
3	Sassy
4	Bonnie
5	Clyde
6	Nadia

MY_PETS PL/SQL Table

Referencing an item in a table – tablename(item_key)

e.g. current_dog := MY_PETS(2);

Creating PL/SQL tables

User-defined data subtype:

```
TYPE table_name  IS TABLE OF item_data_type
INDEX BY BINARY_INTEGER;
```

Declaring a variable of type user-defined subtype:

```
Variable_name        table_name;
```

Example:

```
DECLARE
        TYPE  pet_names_table  IS TABLE OF VARCHAR2(30)
        INDEXED BY BINARY_INTEGER;

        My_pets        PET_NAMES_TABLE;
```

Inserting values in a PL/SQL table:

Table_variable_name(item_index_value) := item_value;

My_pets(1) := ‘Shadow’;

```
DECLARE
        -- declare the user-defined subtype for the table

        TYPE  pet_names_table  IS TABLE OF  VARCHAR2(30)
        INDEXED BY BINARY_INTEGER;

        -- declare the table based on the user-defined subtype

        my_pets         PET_NAMES_TABLE;
BEGIN
        -- populate the table

        my_pets(1) := ‘Shadow’;
        my_pets(2) := ‘Dusty’;
        my_pets(3) := ‘Sassy’;
        my_pets(4) := ‘Bonnie’;
        my_pets(5) := ‘Clyde’;
        my_pets(6) := ‘Nadia’;

        DBMS_OUTPUT.PUT_LINE(‘The first value in the MY_PETS table is ‘ ||
        my_pets(1));
END;
/
```

The first value in the MY_PETS table is Shadow

Dynamically Populating a PL/SQL Table Using Database Values

PL/SQL tables can be populated from database tables using cursor to retrieve values.

```
DECLARE
        -- declare the PL/SQL table
        TYPE item_table IS TABLE OF item.item_desc%TYPE
        INDEX BY BINARY_INTEGER;

        Current_items_table   ITEM_TABLE;
        -- declare cursor to retieve ITEM_DESC values
        CURSOR item_cursor IS
                SELECT item_id, item_desc
                FROM item;

        Item_cursor_row        item_cursor%ROWTYPE;
BEGIN
        -- populate the table using the cursor
        FOR item_cursor_row IN item_cursor
        LOOP
                Current_items_table(item_cursor_row.item_id) :=
                                                        item_cursor_row.item_desc;
        END LOOP;
END;
/
```

PL/SQL procedure successfully completed.

PLSQL Table Attributes

Attribute	Description	Examples	Results
COUNT	Returns the number of rows in the table	My_pets.COUNT	6
DELETE(row_key) DELETE(first_key, last_key)	Deletes all table rows, a specified table row, or a range of rows	My_pets.DELETE My_pets.DELETE(1) My_pets.DELETE(1,3)	Deletes all table rows Deletes the row associated with with index 1 Deletes the rows associated with index values 1 though 3
EXISTS(row_key)	Used in conditional statements to return the BOOLEAN value TRUE if the specified row exists and FALSE if the specified row does not exist	IF my_pets.EXISTS(1)	Returns TRUE if the item associated with key value 1 exists
FIRST	Returns the value of the key of the first item in the table	My_pets.FIRST	1
LAST	Returns the value of the key of the last item in the table	My_pets.LAST	6
NEXT(row_key)	Returns the value of the key of the next row after the specified row	My_pets.NEXT(3)	If current row key = 3, returns 4
PRIOR(row_key)	Returns the value of the key of the row immediately before the specified row	My_pets.PRIOR(3)	If current row key = 3, returns 2

Using PL/SQL table attributes to display table values

```
DECLARE
        -- declare the PL/SQL table
        TYPE item_desc_table IS TABLE OF item.item_desc%TYPE
        INDEX BY BINARY_INTEGER;

        Current_items_desc_table      ITEM_DESC_TABLE;
        -- declare cursor to retieve ITEM_DESC values
        CURSOR item_cursor IS
                SELECT item_id, item_desc
                FROM item;

        Item_cursor_row       item_cursor%ROWTYPE;
        Current_table_key     BINARY_INTEGER;
BEGIN
        -- populate the table using the cursor
        FOR item_cursor_row IN item_cursor
        LOOP
                Current_items__desc_table(item_cursor_row.item_id) :=
                                                item_cursor_row.item_desc;
        END LOOP;

        DBMS_OUTPUT.PUT_LINE('The total number of rows in the
            CURRENT_ITEM_DESC_TABLE is' || current_item_desc_table.COUNT);

        -- loop to step through table and list each value
        current_table_key := current_item_desc_table.FIRST;
        LOOP
                DBMS_OUTPUT.PUT_LINE(
                                        current_item_desc_table(current_table_key));
                EXIT WHEN current_table_key = current_item_desc_table.LAST;
                Current_table_key := current_item_desc_table .NEXT(
                                                                Current_table_key );
        END LOOP;
END;
/
```

```
The total number of rows in the CURRENT_ITEM_DESC_TABLE is 5
Men's Expedition Parka
3-Season Tent
Women's Hiking Shorts
Women's Fleece Pullover
Children's Breachcomber Sandals
```

Creating a PL/SQL Table of Records

Table of records: PL/SQL table that stores multiple data values that are referenced by a unique key.

By storing the information in a PL/SQL table, the program does not have to repeatedly query the database, which improves processing performance.

```
DECLARE
        TYPE  table_data_type_name   IS  TABLE OF
                database_table_name%ROWTYPE
        INDEX BY BINARY_INTEGER;
```

```
DECLARE
        TYPE  item_table IS TABLE OF item%ROWTYPE
        INDEX BY BINARY_INTEGER;
        Current_item_table   TABLE_ITEM;
```

To assign a value to a specific field in a table of records:

```
Table_name(key_value).database_fieldname := field_value;
```

```
CURRENT-ITEM_TABLE(1).item_desc := 'Women''s Hiking Shorts';
```

```
DECLARE
        -- declare the PL/SQL table of records
        TYPE item_table IS TABLE OF item%ROWTYPE
        INDEX BY BINARY_INTEGER;

        Current_item_table    ITEM_TABLE;
        -- declare cursor to retrieve ITEM values
        CURSOR item_cursor IS
                SELECT * FROM item;

        Item_cursor_row       item_cursor%ROWTYPE;
        Current_table_key     BINARY_INTEGER;

BEGIN
        -- populate the table using the cursor
        FOR item_cursor_row IN item_cursor
        LOOP
                Current_table_key := item_cursor_row.item_id;
                Current_item_table(current_table_key).item_id :=
                                                item_cursor_row.item_id;
                Current_item_table(current_table_key).item_desc :=
                                                item_cursor_row.item_desc;
                Current_item_table(current_table_key).category_id :=
                                                item_cursor_row.category_id;
                Current_item_table(current_table_key).item_image:=
                                                item_cursor_row.item_image;
        END LOOP;
        -- list ITEM_ID and ITEM_DESC values of CURRENT_ITEM_TABLE rows
        current_table-key := current_item_table.FIRST;
        LOOP
                DBMS_OUTPUT.PUTLINE(current_item_table(current_table_key).
                  Item_id || '   ' || current_item_table(current_table_key).item_desc);
                EXIT WHEN current_table_key = current_item_table.LAST;
                Current_table_key := current_item_table.NEXT(current_table_key);
        END LOOP;
END;
/
```

```
559     Men's Expedition Parka
786     3-Season Tent
894     Women's Hiking Shorts
897     Women's Fleece Pullover
995     Children's Breachcomber Sandals
```

PL/SQL procedure successfully completed.

Exception Handling

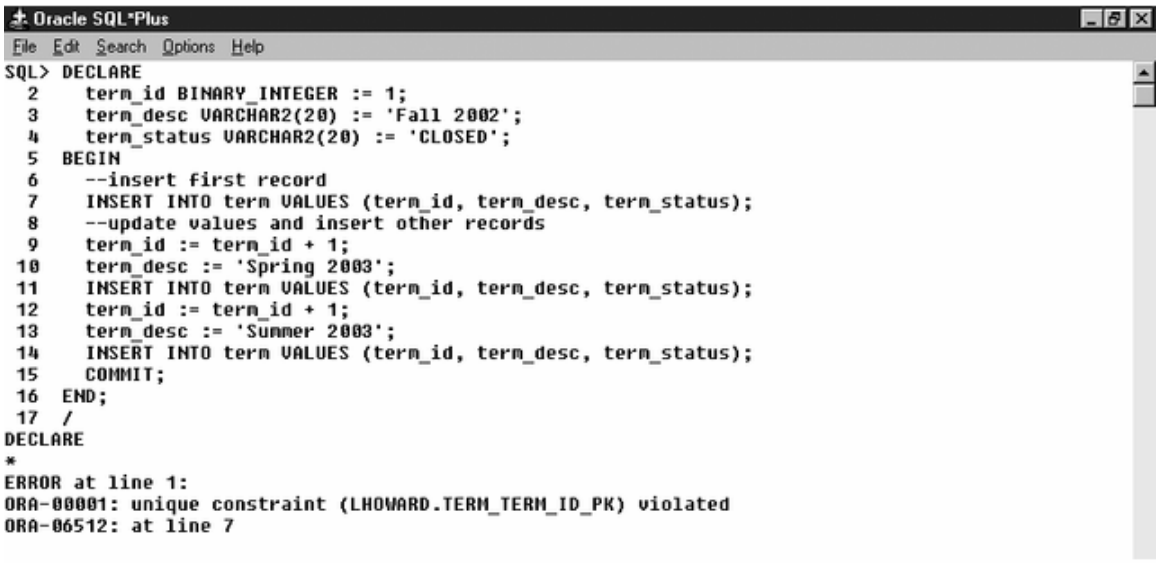


Fig. 4-49: Example of a run-time error. Violation of primary key constraint.

When a run-time error occurs, an **exception**, or unwanted event, is **raised**. Three kinds of exceptions: **predefined**, **undefined**, and **user-defined.**

Predefined Exceptions

Oracle Error Code	Exception Name	Description
ORA-00001	DUP_VAL_ON_INDEX	Unique constraint on primary key violated
ORA-01001	INVALID_CURSOR	Illegal cursor operation
ORA-01403	NO_DATA_FOUND	Query returns no records
ORA-01422	TOO_MANY_ROWS	Query returns more rows than anticipated
ORA-01476	ZERO_DIVIDE	Division by zero
ORA-01722	INVALID_NUMBER	Invalid number conversion (like trying to convert ‘2B’ to a number
ORA-06502	VALUE_ERROR	Error in truncation, arithmetic, or conversion operation

User can display custom error messages for predefined exceptions. The general format for handling predefined and other exceptions is:

```
EXCEPTION
        WHEN  <exception1 name.  THEN
                <exception handling statements>;
         WHEN  <exception1 name.  THEN
                <exception handling statements>;
…
        WHEN  others  THEN
                <exception handling statements>;
END;
```

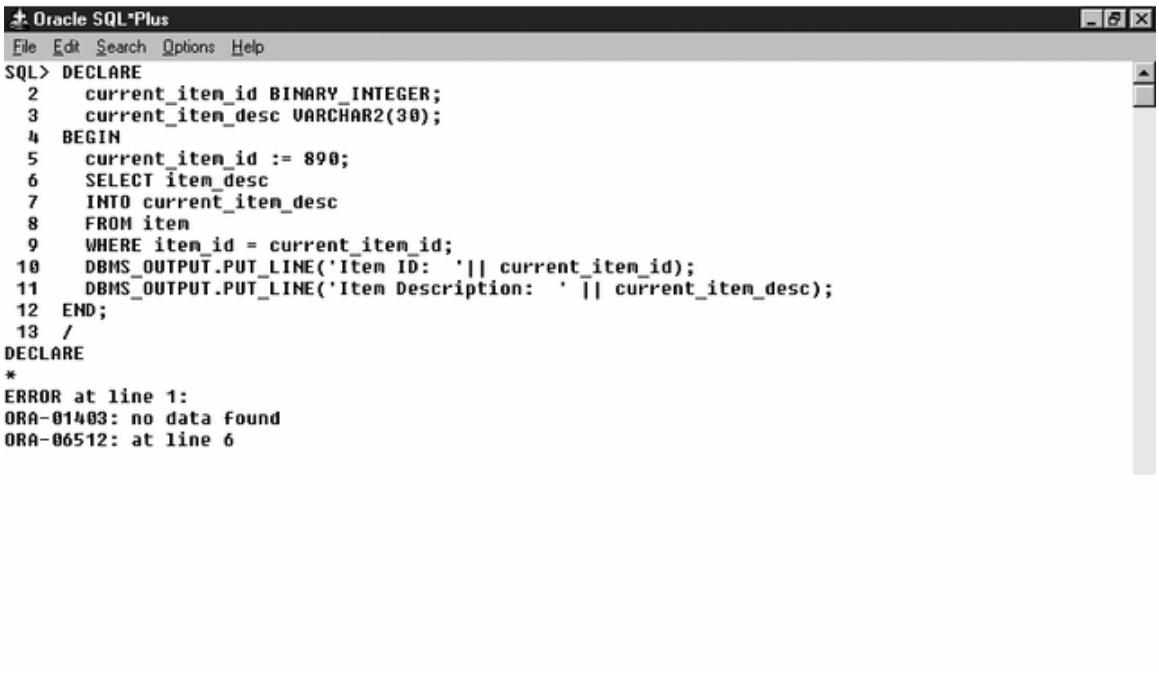


Figure 4-50: PL/SQL program with predefined exception

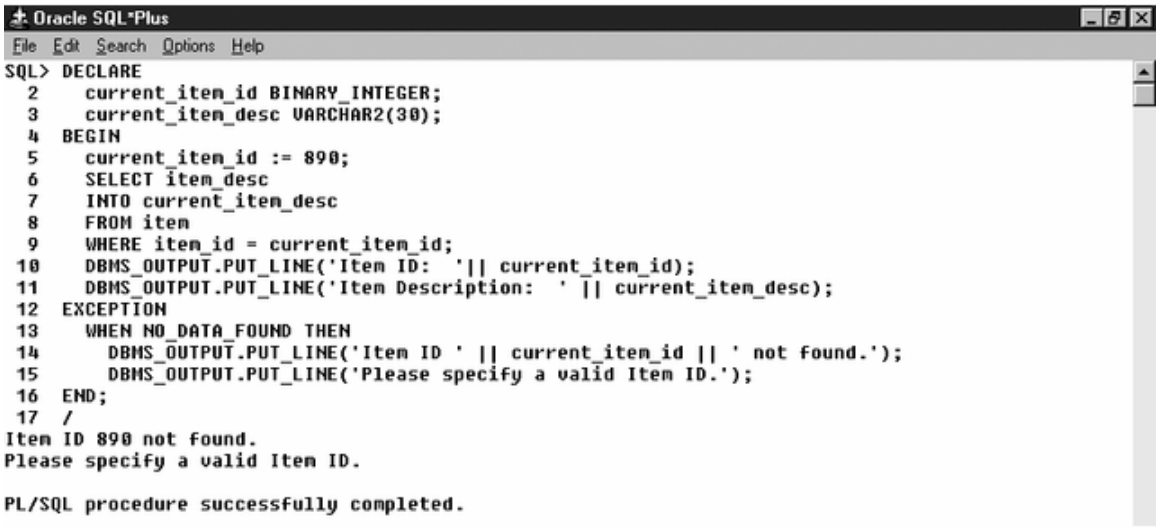


Figure 4-51: PL/SQL program that handles a predefined exception

During program development, it is helpful to use the WHEN OTHERS exception handler to display the associated Oracle error code number and error message for unanticipated errors.

To do this, you use the **SQLERRM** function. This function returns a character string that contains the Oracle error code and the text of the error code's error message for the most recent Oracle error generated.

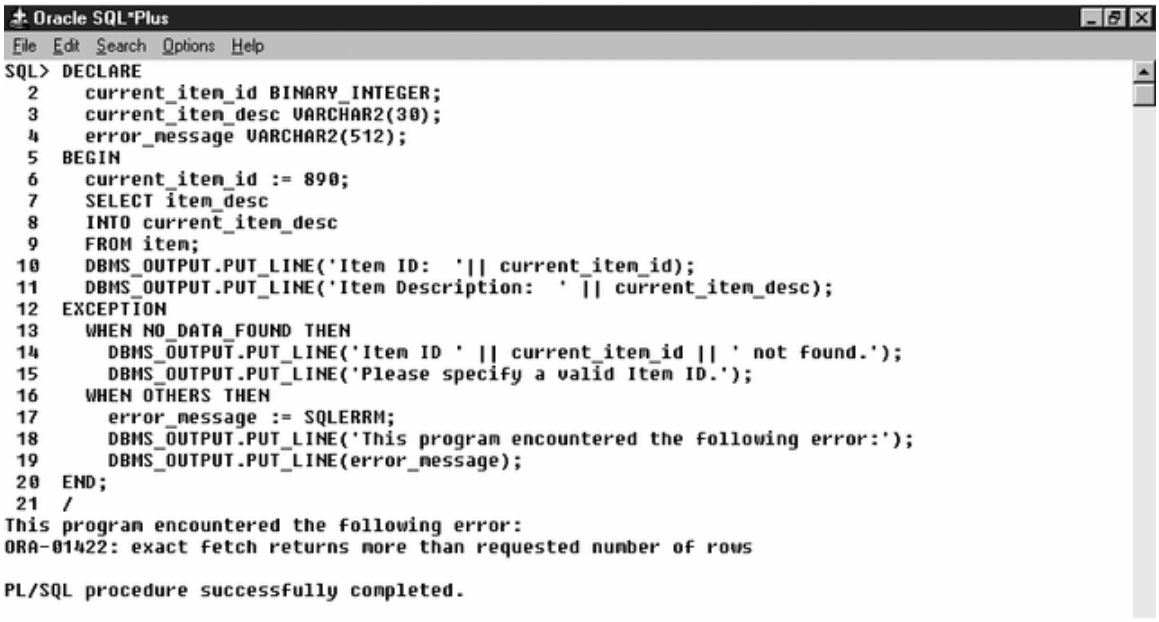


Fig. 4-52: Displaying the Oracle error code and message in the WHEN OTHERS exception handler.

Undefined Exceptions

- less common errors that have not been given an explicit exception name.
- In the following example, a program attempts to insert a record into the TERM table where the data value for the third table field (STATUS) is NULL.
- The TERM table has a NOT NULL constraint for that field.

```
DECLARE
        Term_id         BINARY_INTEGER;
        Term_desc       VARCHAR2(20);
BEGIN
        Term_id := 7;
        Term_desc := 'Fall 2002';
        -- insert the record
        INSERT INTO term VALUES(term_id, term_desc,NULL);
        COMMIT;
END;
/

DECLARE
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("LHOWARD"."TERM"."STATUS")
ORA-06512: at line 8
```

PL/SQL program with undefined exception

To handle an undefined exception, you must explicitly declare the exception in the DECLARE section of the program and associate it with a specific Oracle error code.

```
DECLARE
        <e_exception name> EXCEPTION;
        PRAGMA     EXCEPTION_INIT(<e_exception name>,<Oracle error code>);
```

The PRAGMA EXCEPTION_INIT command tells the compiler to associate the given exception name with a specific Oracle error code.

```
DECLARE
        Term_id         BINARY_INTEGER;
        Term_desc       VARCHAR2(20);

        E_not_null_insert       EXCEPTION;
        PRAGMA      EXCEPTION_INIT(e_not_null_insert, -1400)
BEGIN
        Term_id := 7;
        Term_desc := 'Fall 2002';
        -- insert the record
        INSERT INTO term VALUES(term_id, term_desc,NULL);
        COMMIT;
EXCEPTION
        WHEN e_not_null_insert THEN
                DBMS_OUTPUT.PUT_LINE('You must specify a data value for all
                                                                TERM fields');
END;
/
```

You must specify a data value for all TERM fields

User-Defined Exceptions

User-defined exceptions are used to handle exceptions that will not cause an Oracle run-time error, but require exception handling to enforce business rules or to ensure the integrity of the database. The general format is:

```
DECLARE
        <e_exception name> EXCEPTION;
        <other variable declarations>;
BEGIN
        <other program statements>
        IF <undesirable condition> THEN
                RAISE <e_exception name>
        END IF;
        <other program statements>;
EXCEPTION
        <e_exception name>
                <error-handling statements>;
END;
```

```
DECLARE
        Current_s_id            BINARY_INTEGER := 100;
        Current_c_sec_id        BINARY_INTEGER :=1000;
        Current_grade           CHAR(1);
        E_null_grade_delete     EXCEPTION;
BEGIN
        SELECT grade
        INTO current_grade
        FROM enrollment
        WHERE s_id = current_s_id
        AND c_sec_id = current_c_sec_id;

        IF current_grade IS NULL THEN
                DELETE FROM enrollment
                WHERE s_id = 100 AND c_sec_id = 1000;
        ELSE
                RAISE e_null_grade_delete;
        END IF;
EXCEPTION
        WHEN e_null_grade_delete THEN
                DBMS_OUTPUT.PUT_LINE('The grade field for the current enrollment
record is not null.');
                DBMS_OUTPUT.PUT_LINE('Enrollment record not deleted.');
        END;
/
```

The grade field for the current enrollment record is not null.
Enrollment record not deleted.

Nested PL/SQL Program Blocks

One program block contains another program block. A PL/SQL program can contain multiple nested program blocks.

```
DECLARE
        -- outer block
        current_bldg_code      VARCHAR2(8) := 'BUS';
BEGIN
        DECLARE
                -- inner block
                CURSOR bldg_cursor IS
                        SELECT *
                        FROM location
                        WHERE bldg_code = current_bldg_code;
                Bldg_cursor_row        bldg_cursor%ROWTYPE;
        BEGIN
                FOR bldg_cursor_row IN bldg_cursor
                LOOP
                        DBMS_OUTPUT.PUT_LINE(bldg_cursor_row.bldg_code || '  ' ||
                                                                bldg_cursor_row.room);
                END LOOP;
        END;
END;
/
```

```
BUS             185
BUS             484
BUS             421
BUS             211
BUS             424
BUS             402
BUS             433
```

In some cases a variable declared in an outer block will not be visible in a nested block. This happens when a variable in an inner block is given the same name as a variable in outer block.

```
DECLARE
        -- outer block
        current_bldg_code      VARCHAR2(8) := 'BUS';
BEGIN
        DECLARE
                -- inner block
                current_bldg_code      VARCHAR2(8) := 'CR';

                CURSOR bldg_cursor IS
                        SELECT *
                        FROM location
                        WHERE bldg_code = current_bldg_code;
                Bldg_cursor_row        bldg_cursor%ROWTYPE;
        BEGIN
                FOR bldg_cursor_row IN bldg_cursor
                LOOP
                        DBMS_OUTPUT.PUT_LINE(bldg_cursor_row.bldg_code || '  ' ||
                                                                bldg_cursor_row.room);
                END LOOP;
        END;
END;
/

CR      101
CR      202
CR      103
CR      105
```

Exception Handling in Nested Program Blocks

One of the main reasons for creating nested blocks is to facilitate exception handling. If an exception is raised in an inner block, program control resumes in the outer block.

Sometimes you need to write a program in which an exception will be raised in some situations but program execution should still continue.

Example: suppose that the capacity of some of the classrooms at Northwoods University changes, and we want to determine which course section has exceeded the capacity of its rooms. An exception will be raised when a classroom cannot accommodate the maximum enrollment in a course section, and an error message will appear. Then the program execution will continue and examine the maximum enrollment and capacity of the next course section.

First update the capacity field for LOC_ID 49 to raise an exception.

```
UPDATE location
SET capacity = 30
WHERE loc_id =49;
COMMIT;


DECLARE
        CURSOR c-sec_cursor IS
                SELECT c_sec_id, location.loc_id, max_enrl, capacity
                FROM course_section, location
                WHERE location.loc_id = course_section.loc_id;
        C_sec_row      c_sec_cursor%ROWTYPE;
BEGIN
        FOR c_sec_row IN c_sec_cursor
        LOOP
                -- inner block
                DECLARE
                        E_capacity_error        EXCEPTION;
                BEGIN
                        IF c_sec_row.max_enrl > c_sec_row.capacity THEN
                                RAISE e_capacity_error;
                        END IF;
                -- exception handler
                EXCEPTION
                        WHEN e_capacity_error THEN
                                DBMS_OUTPUT.PUT_LINE('Capacity error in c_sec_id'
                                                                        || c_sec_row.c_sec_id);
                END;
        END LOOP;
END;
/
```

```
Capacity error in c_sec_id 1008
Capacity error in c_sec_id 1009
Capacity error in c_sec_id 1012
```

Suppose that an exception is raised in an inner block, but no exception handler exists for this exception in the exception handler for the inner block. However, an exception handler for this particular exception exists in the EXCEPTION section of the outer block. The outer block's exception handler will handle the exception, but the program will immediately terminate.

```
DECLARE
        CURSOR c-sec_cursor IS
                SELECT c_sec_id, location.loc_id, max_enrl, capacity
                FROM course_section, location
                WHERE location.loc_id = course_section.loc_id;
        C_sec_row      c_sec_cursor%ROWTYPE;
        Current_c_sec_id       NUMBER(6);
        e_capacity_error       EXCEPTION;
BEGIN
        FOR c_sec_row IN c_sec_cursor
        LOOP
                -- inner block
                BEGIN
                        IF c_sec_row.max_enrl > c_sec_row.capacity THEN
                                Current_c_sec_id := c_sec_row.c_sec_id;
                                RAISE e_capacity_error;
                        END IF;
                END;
        END LOOP;
EXCEPTION
        WHEN e_capacity_error THEN
                DBMS_OUTPUT.PUT_LINE('Capacity error in c_sec_id'
                                                        || c_sec_row.c_sec_id);

END;
/
```

Capacity error in c_sec_id 1008