PL/SQL

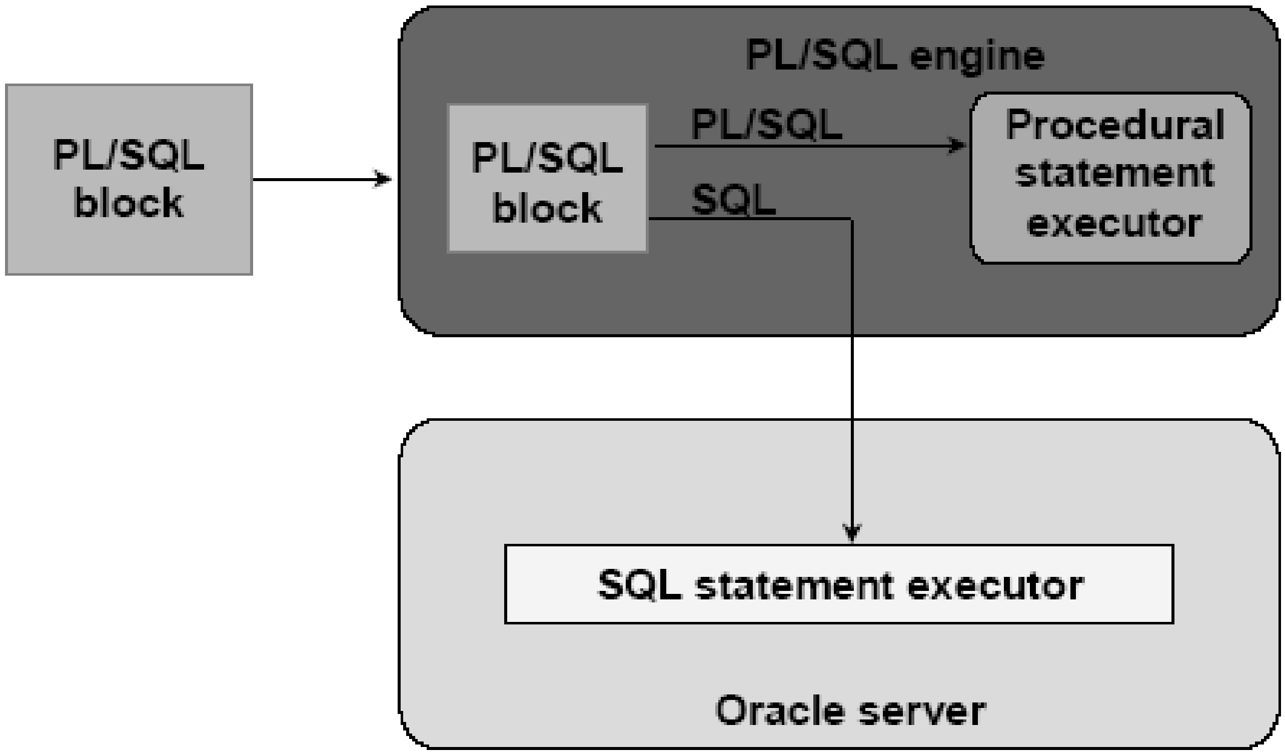
PL/SQL

- Introduction
- Structure of a block
- Variables and types
- Accessing the database
- Control flow
- Cursors
- Exceptions
- Procedures and functions

Introduction

- PL/SQL stands for Procedural Language/SQL.
- PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL.
- The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other. Typically, each block performs a logical action in the program.

PL/SQL Environment



Benefits

- Integrate of database technology and procedural programming capabilities
- Provide improved performance of an application
- Modularize program development
- Portable among host environments supporting Oracle server and PL/SQL
- Handle errors

Structure of a Block

```
DECLARE
 /* Declarative section: variables, types, cursors,
  user-defined exceptions */
BEGIN
/* Executable section: procedural and SQL statements
  go here. */
 /* This is the only section of the block that is
  required. */
EXCEPTION
/* Exception handling section: error handling
  statements go here. */
END; /* mandatory */
```

Block (1) (Declare)

- The only SQL statements allowed in a PL/SQL program are SELECT, INSERT, UPDATE, DELETE and several other data manipulation statements plus some transaction control.
- Data definition statements like CREATE, DROP, or ALTER are not allowed.
- The executable section also contains constructs such as assignments, branches, loops, procedure calls, and triggers

Block (2)

- PL/SQL is not case sensitive. C style comments (/* ... */) may be used.

- To execute a PL/SQL program, we must follow the program text itself by: a line with a single dot ("."), and then a line with “run”; or a line with “/”

- We can invoke a PL/SQL program either by typing it in sqlplus or by putting the code in a file and invoking the file

Variables and Types (1)

- Information is transmitted between a PL/SQL program and the database through **variables**.
- Every variable has a specific **type** associated with it, that can be:
  - One of the types used by SQL for database columns
  - A generic type used in PL/SQL such as NUMBER
  - Declared to be the same as the type of some database column

Variables and Types (2)

- The most commonly used generic type is NUMBER. Variables of type NUMBER can hold either an integer or a real number.
- The most commonly used character string type is VARCHAR(*n*), where *n* is the maximum length of the string in bytes. This length is required, and there is no default.

Example

```
DECLARE
   price  NUMBER;
   myBeer VARCHAR(20);
```

Declaring Boolean variables

- Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.
- The variables are compared by the logical

operators AND, OR, and NOT.

- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

Example

```
DECLARE
   v_flag BOOLEAN := FALSE;
BEGIN
   v_flag := TRUE;
END;
```

The %TYPE attribute

- A PL/SQL variable can be used to manipulate data stored in a existing relation.
- The variable must have the same type as the relation column. If there is any type mismatch, variable assignments and comparisons may not work the way you expect.
- To be safe, instead of hard coding the type of a variable, you should use the %TYPE operator.

Ex:

table name column name

```
DECLARE        myBeer Beers.name%TYPE;
```

gives PL/SQL variable myBeer whatever type was declared for the name column in relation Beers.

The %ROWTYPE attribute

- A variable may also have a type that is a record with several fields.
- The simplest way to declare such a variable is to use %ROWTYPE on a relation name. The result is a record type in which the fields have the same names and types as the attributes of the relation.

Ex:

```
DECLARE        beerTuple Beers%ROWTYPE;
```

makes variable beerTuple be a record with fields name and manufacture, assuming that the relation has the schema Beers(name, manufacture).

Default values and assignments

- The initial value of any variable, regardless of its type, is NULL.
- We can assign values to variables, using the ":=" operator.
- The assignment can occur either immediately after the type of the variable is declared, or anywhere in the executable portion of the program.

Ex:

```
DECLARE
 a NUMBER := 3;
BEGIN
 a := a + 1;
END;
.
run;
```

Bind variables (see example)

- Variable that you declare in a host environment (ex: SQL*Plus), used to pass run-time values (number or characters) into or out of PL/SQL programs
- The only kind that may be printed with a print command.
- Bind variables must be prefixed with a colon in PL/SQL statements

Steps to create a bind variable

1. We declare a bind variable as follows:
   VARIABLE <name> <type>
   where the type can be only one of three things: NUMBER, CHAR, or CHAR(*n*).
2. We may then assign to the variable in a following PL/SQL statement, but we must prefix it with a colon.
3. Finally, we can execute a statement :

PRINT :<name>;

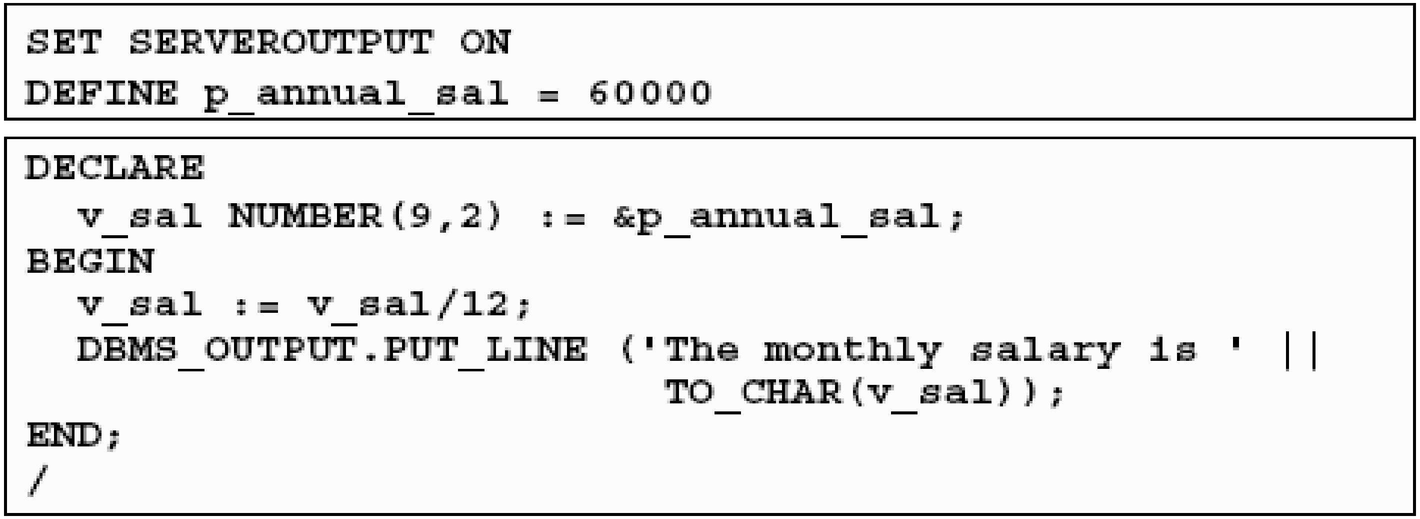
outside the PL/SQL statement

Example

```
VARIABLE
 x NUMBER
BEGIN
   x := 1;
END;
.
run;
PRINT :x;
```

DBMS_OUTPUT.PUT_LINE package

- An Oracle-supplied packaged procedure
- An alternative for displaying data from a PL/SQL block
- Must be enabled in SQL*Plus with: **`SET SERVEROUTPUT ON`**



```
SET SERVEROUTPUT ON
DEFINE p_annual_sal = 60000
```

```
DECLARE
   v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
   v_sal := v_sal/12;
   DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' ||
                         TO_CHAR(v_sal));
END;
/
```

Simple programs accessing the database

- The simplest form of program has some declarations followed by an executable section consisting of one or more of the SQL statements with which we are familiar.
- After the SELECT clause, we must have an INTO clause listing variables, one for each attribute in the SELECT clause, into which the components of the retrieved tuple must be placed.
- The SELECT statement in PL/SQL only works if the result of the query contains a single row. If the query returns more than one rows, you need to use a cursor

Example (SQL)

```
CREATE TABLE T1(
       e INTEGER,
       f INTEGER );

DELETE FROM T1;

INSERT INTO T1 VALUES(1, 3);
INSERT INTO T1 VALUES(2, 4);
```

e	f
1	3
2	4

Example (PL/SQL)

e	f
1	3
2	4

```
DECLARE
      a NUMBER;
      b NUMBER;
BEGIN
   SELECT e,f INTO a,b FROM T1 WHERE
   e>1;
   INSERT INTO T1 VALUES(b,a);
END;
  .
run;
```

a=2, b=4

e	f
1	3
2	4
4	2

Control flow

- PL/SQL allows you to branch and create loops in a fairly familiar way.
- An IF statement looks like:

```
IF <condition> THEN <statement_list>
ELSE <statement_list>
END IF;
```

- The ELSE part is optional. If you want a multiway branch, use:

```
IF <condition_1> THEN ...
ELSIF <condition_2> THEN ... ... ...
ELSIF <condition_n> THEN ...
ELSE ...
END IF;
```

Example

e	f
1	3
2	4

```
DECLARE
 a NUMBER;
 b NUMBER;
BEGIN
    SELECT e,f INTO a,b FROM T1 WHERE e>1;
    IF b=1 THEN
        INSERT INTO T1 VALUES (b,a);
    ELSE
        INSERT INTO T1 VALUES (b+10,a+10);
    END IF;
END;
.
run;
```

a=2, b=4

e	f
1	3
2	4
14	12

Loops

- Loops are created with the following:

```
LOOP
  <loop_body> /* A list of statements. */
 END LOOP;
```

- At least one of the statements in <loop_body> should be an EXIT statement of the form

```
EXIT WHEN <condition>;
```

- The loop breaks if <condition> is true.

Example

```
DECLARE
  i NUMBER := 1;
BEGIN
   LOOP
      INSERT INTO T1 VALUES (i,i);
           i  := i+1;
      EXIT WHEN i>100;
   END LOOP;
END;
 .
run;
```

Other useful loop-forming statements

- A WHILE loop can be formed with:

```
 WHILE <condition>
  LOOP
     <loop_body: Action>
  END LOOP;
```

- A simple FOR loop can be formed with:

```
FOR <var> IN <start>..<finish>
 LOOP
  <loop_body: Action>
 END LOOP;
```

- Here, <var> can be any variable; it is local to the for-loop and need not be declared. Also, <start> and <finish> are constants.

Cursors

- A cursor is a variable that runs through the tuples of some relation.
  - Can be a stored table or the answer to some query.
- By fetching into the cursor each tuple of the relation, we can write a program to read and process the value of each such tuple.
- If the relation is stored, we can also update or delete the tuple at the current cursor position.

Example (1)

Uses T1(e,f) whose tuples are pairs of integers.

The program will delete every tuple whose first component is less than the second, and insert the reverse tuple into T1.

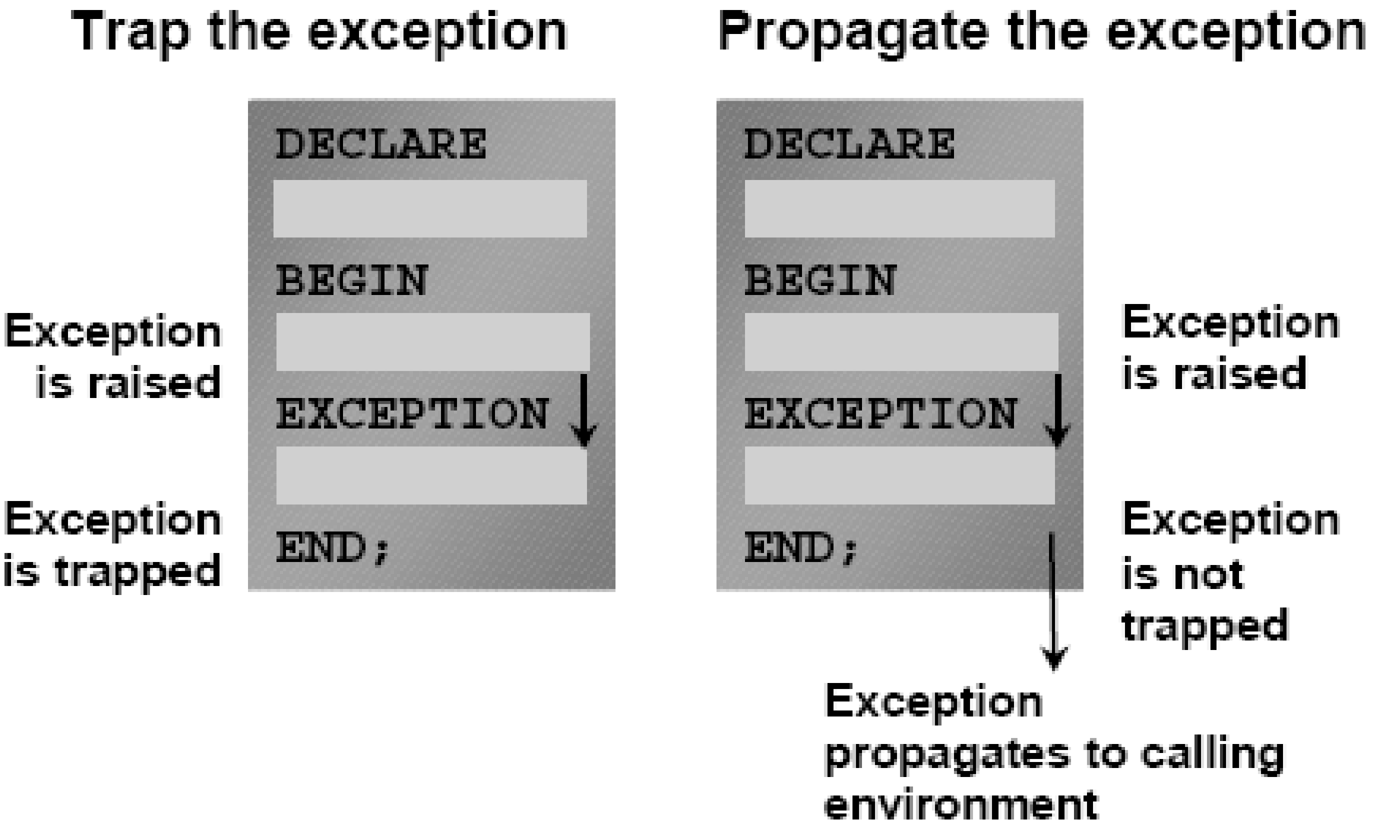
Example (2)

```
DECLARE
     /* Output variables to hold the result of the query: */
     a T1.e%TYPE;
     b T1.f%TYPE;
    /* Cursor declaration: */
     CURSOR T1Cursor IS
         SELECT e, f FROM T1 WHERE e < f
     FOR UPDATE;
BEGIN
    OPEN T1Cursor;
    LOOP  /* Retrieve each row of the result of the above
  query  into PL/SQL variables: */
        FETCH T1Cursor INTO a, b;
       /* If there are no more rows to fetch, exit the loop: */
        EXIT WHEN T1Cursor%NOTFOUND;
        /* Delete the current tuple: */
        DELETE FROM T1 WHERE CURRENT OF T1Cursor;
        /* Insert the reverse tuple: */
        INSERT INTO T1 VALUES(b, a);
     END LOOP;
     /* Free cursor used by the query. */
    CLOSE T1Cursor;
END;
```

Exceptions

- An exception is an identifier in PL/SQL that is raised during execution.
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

Handling exceptions



Trap the exception

DECLARE

BEGIN

Exception is raised

EXCEPTION

Exception is trapped

END;

Propagate the exception

DECLARE

BEGIN

Exception is raised

EXCEPTION

Exception is not trapped

END;

Exception propagates to calling environment

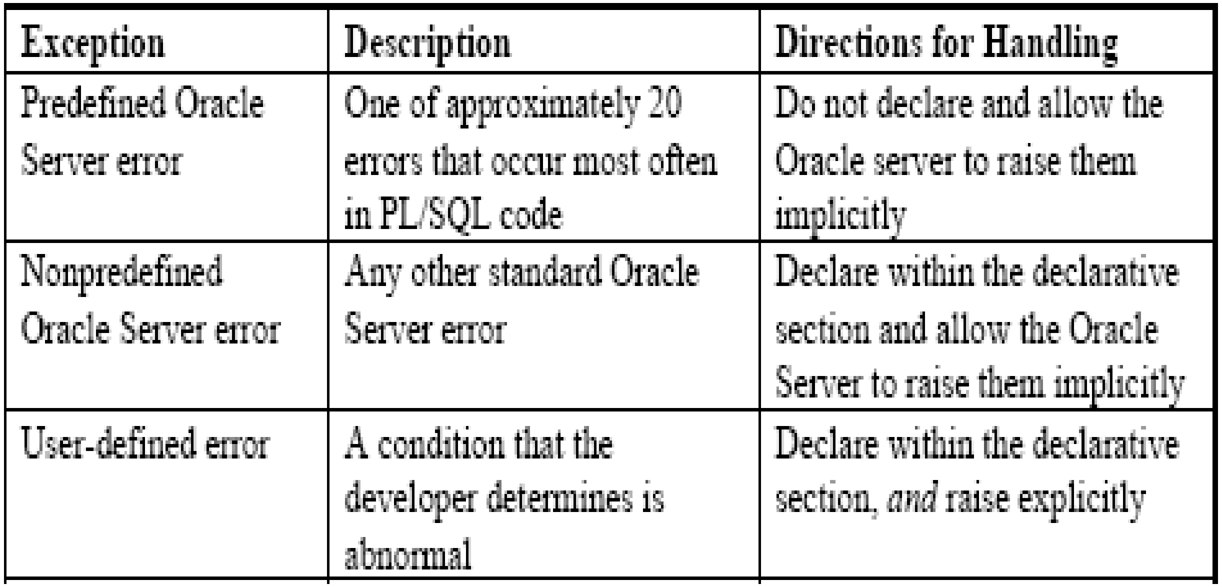
Exception types

- Predefined Oracle Server
- Nonpredefined Oracle Server

Implicitly raised

- User-defined

Explicitly raised



Exception	Description	Directions for Handling
Predefined Oracle Server error	One of approximately 20 errors that occur most often in PL/SQL code	Do not declare and allow the Oracle server to raise them implicitly
Nonpredefined Oracle Server error	Any other standard Oracle Server error	Declare within the declarative section and allow the Oracle Server to raise them implicitly
User-defined error	A condition that the developer determines is abnormal	Declare within the declarative section, *and* raise explicitly

Trapping exceptions

Syntax:

```
EXCEPTION
WHEN exception1 [OR exception2 . . .] THEN
   statement1; /* one or more PL/SQL or SQL
  statements */
   statement2;
   . . .
 [WHEN exception3 [OR exception4 . . .] THEN
   statement1;
   statement2;
   . . .]
 [WHEN OTHERS THEN
   statement1;
   statement2;
   . . .]
```

- Only one handler is processed
- WHEN OTHERS is unique and must be the last handler

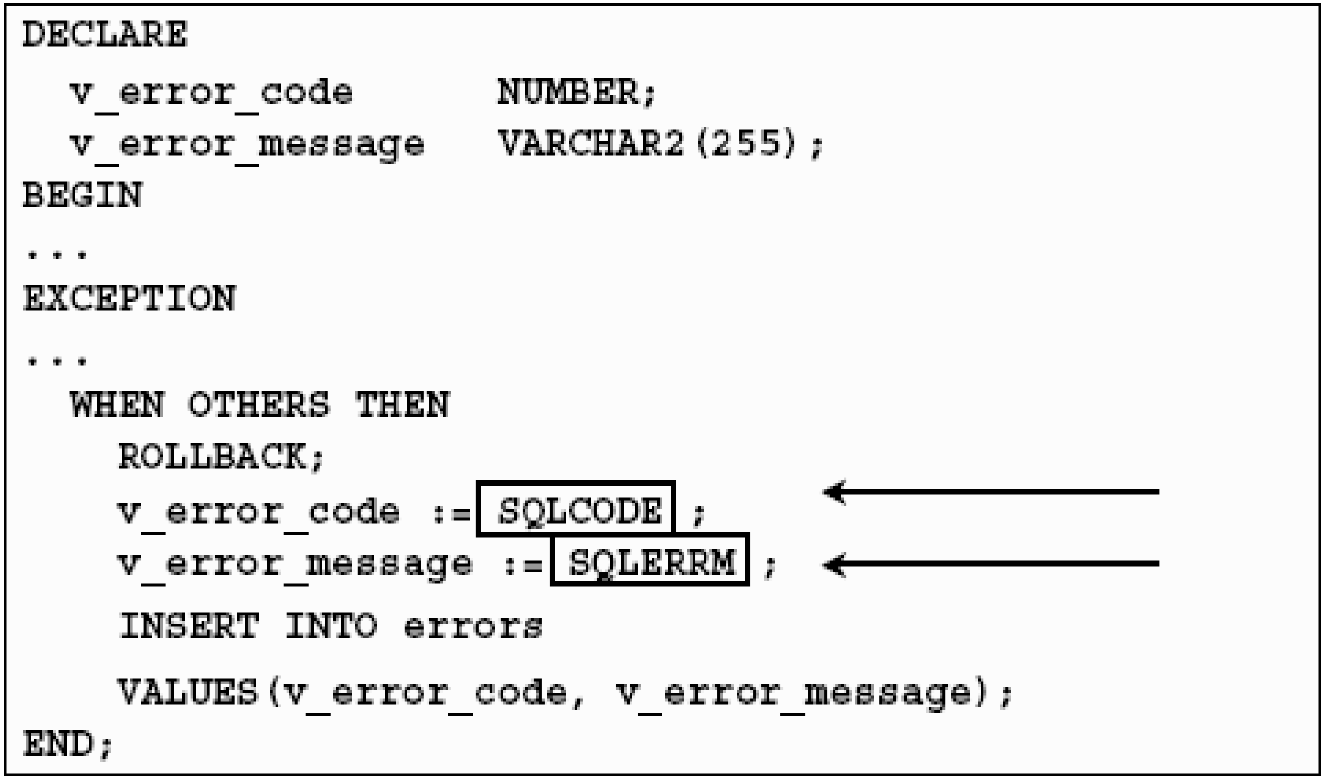
Trapping pre-defined Oracle server errors

- **Reference the standard name in the exception handling routine.**
- **Sample predefined exceptions:**
  - **NO_DATA_FOUND**
  - **TOO_MANY_ROWS**
  - **INVALID_CURSOR**
  - **ZERO_DIVIDE**
  - **DUP_VAL_ON_INDEX**

Functions for Trapping Exceptions

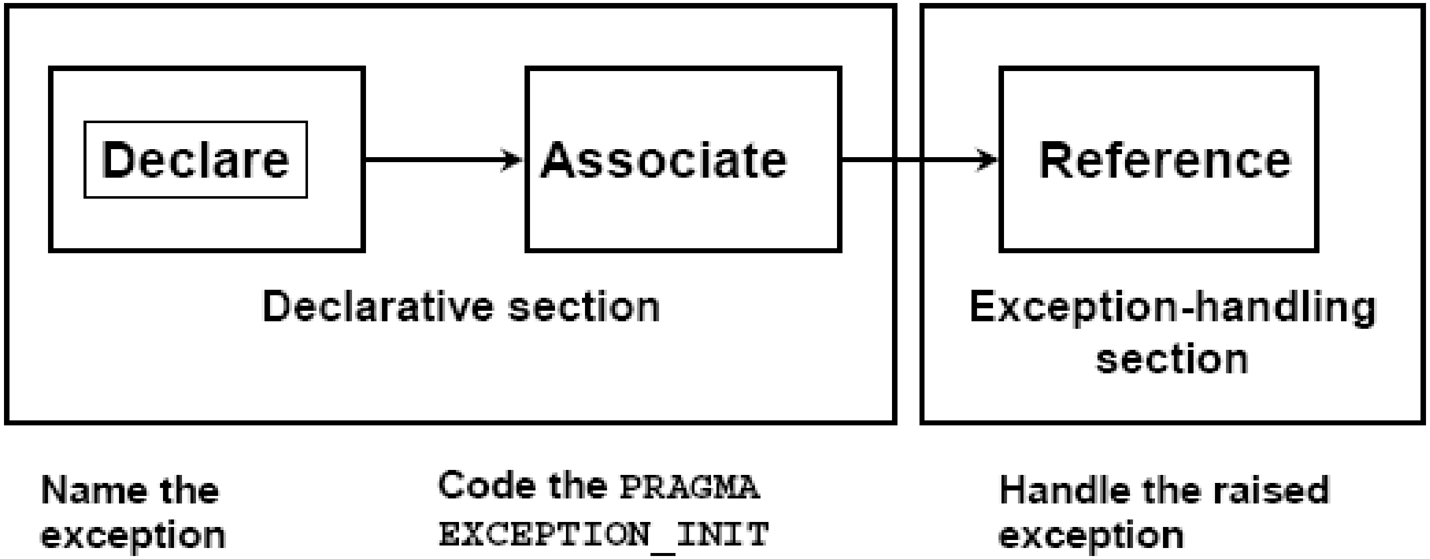
- When an exception occurs, you can identify the associated error code or error message by using two functions.
- Based on the values of the code or message, you can decide which subsequent action to take based on the error.
  - SQLCODE: Returns the numeric value for the error code
  - SQLERRM: Returns the message associated with the error number
- Examples of SQLCODE values:
  - 0: no exception encountered
  - 1: user-defined exception
  - +100: NO_DATA_FOUND_EXCEPTION
  - Negative number: another Oracle server error number

Example



```
DECLARE
  v_error_code        NUMBER;
  v_error_message     VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;
    INSERT INTO errors
    VALUES(v_error_code, v_error_message);
END;
```

Trapping Nonpredefined Oracle Server Errors



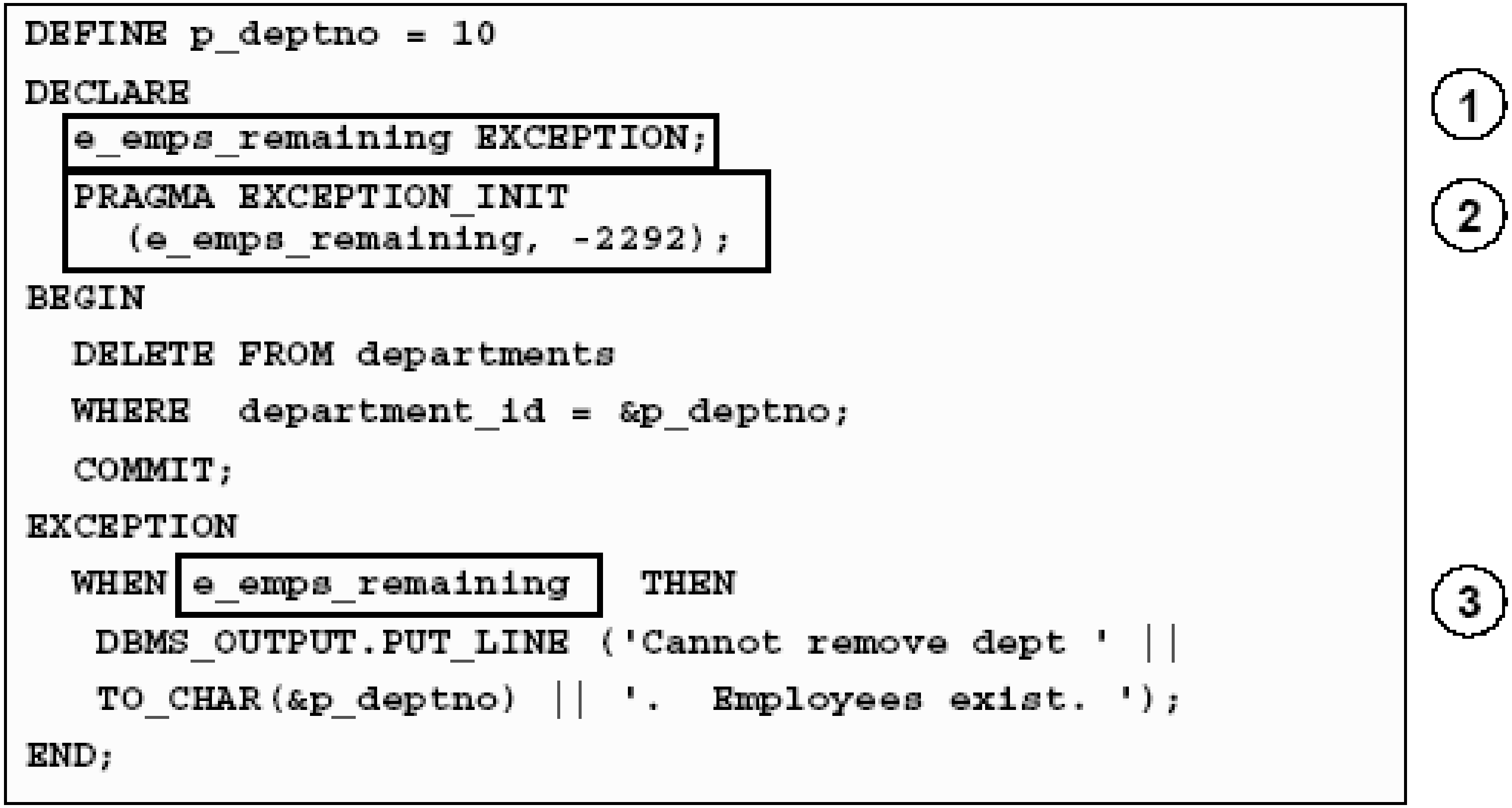
Name the exception

Code the PRAGMA EXCEPTION_INIT

Handle the raised exception

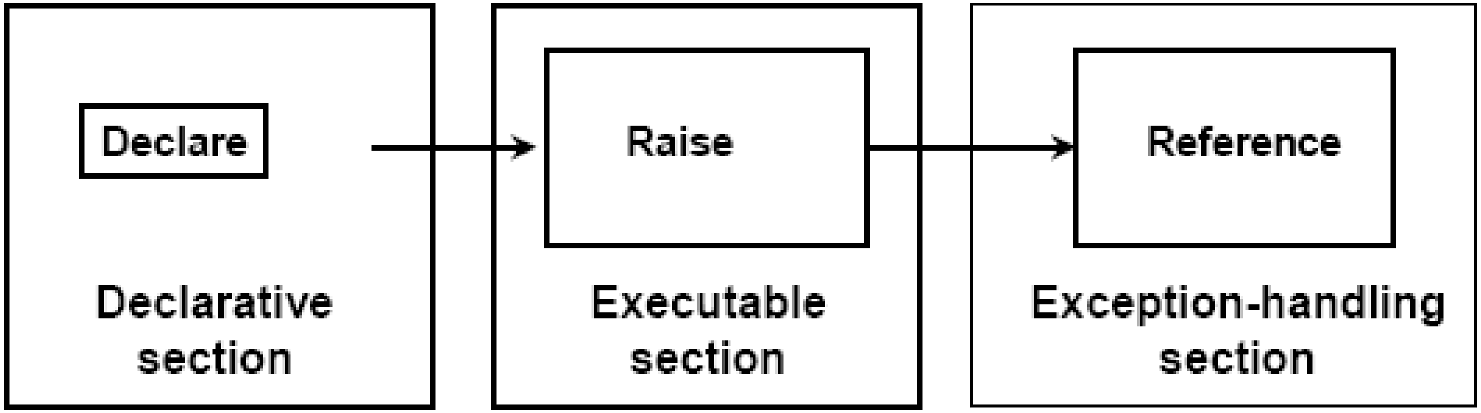
Example

Trap for Oracle server error number –2292, an integrity constraint violation.



```
DEFINE p_deptno = 10
DECLARE
  e_emps_remaining EXCEPTION;
  PRAGMA EXCEPTION_INIT
    (e_emps_remaining, -2292);
BEGIN
  DELETE FROM departments
  WHERE  department_id = &p_deptno;
  COMMIT;
EXCEPTION
  WHEN e_emps_remaining  THEN
   DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
   TO_CHAR(&p_deptno) || '.  Employees exist. ');
END;
```

Trapping User-Defined Exceptions

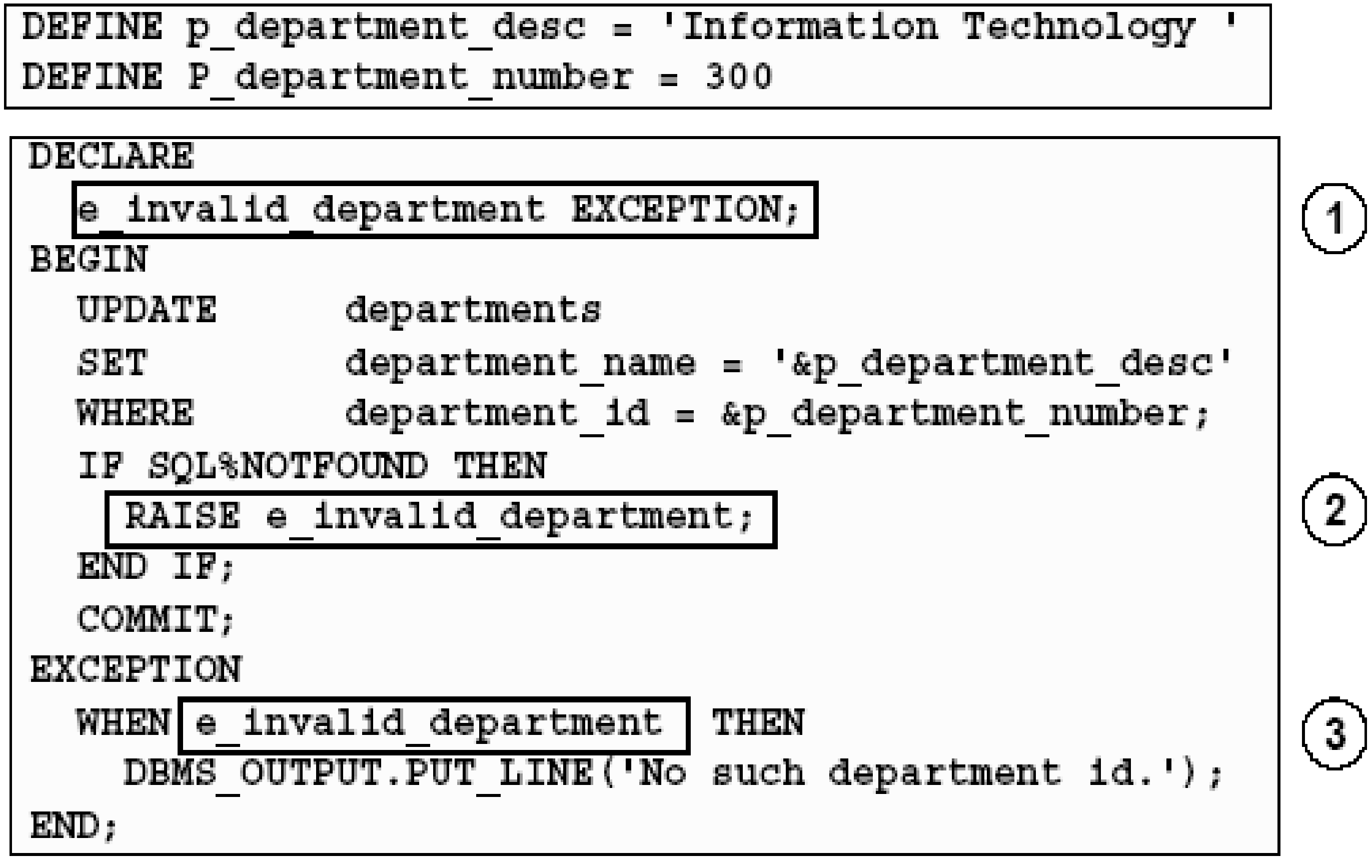


Name the exception.

Explicitly raise the exception by using the `RAISE` statement.

Handle the raised exception.

Example

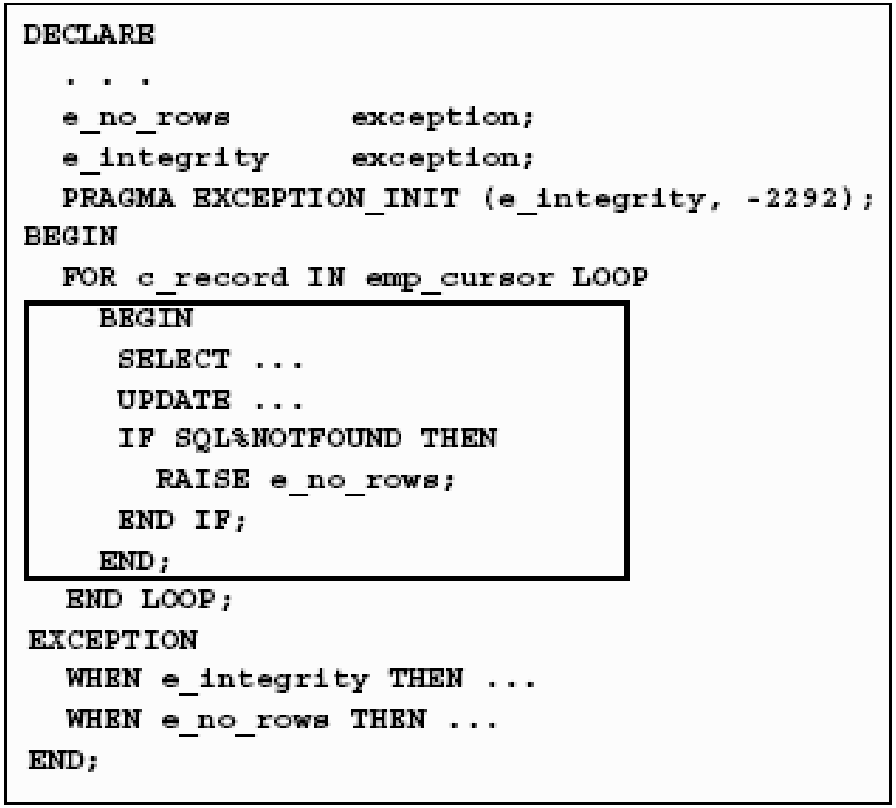


```
DEFINE p_department_desc = 'Information Technology '
DEFINE P_department_number = 300
```

```
DECLARE
  e_invalid_department EXCEPTION;
BEGIN
  UPDATE      departments
  SET         department_name = '&p_department_desc'
  WHERE       department_id = &p_department_number;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_department THEN
    DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
```

Propagating exceptions

Subblocks can handle an exception or pass the exception to the enclosing block.



```
DECLARE
  . . .
  e_no_rows      exception;
  e_integrity    exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP
    BEGIN
     SELECT ...
     UPDATE ...
     IF SQL%NOTFOUND THEN
       RAISE e_no_rows;
     END IF;
    END;
  END LOOP;
EXCEPTION
  WHEN e_integrity THEN ...
  WHEN e_no_rows THEN ...
END;
```

RAISE_APPLICATION_ERROR Procedure

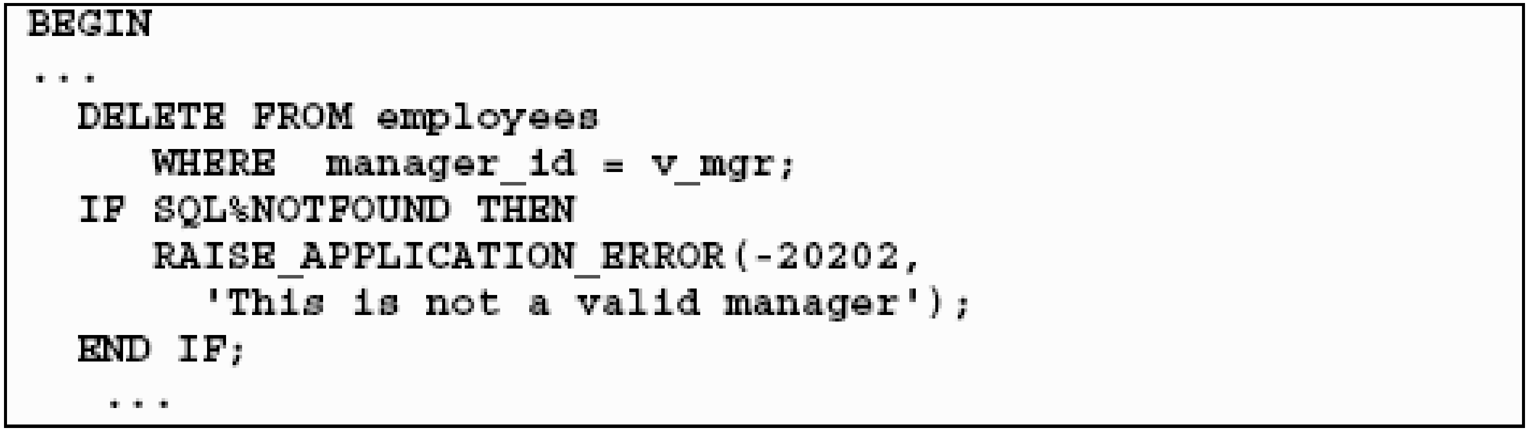
- **Syntax:**

```
raise_application_error (error_number,
message[, {TRUE | FALSE}]);
```

- **You can use this procedure to issue user-error messages from stored subprograms.**
- **You can report errors to your application and avoid returning unhandled exceptions.**
- **Used in two different places:**
  - **Executable section**
  - **Exception section**
- **Returns error conditions to the user in a manner consistent with other Oracle server errors**

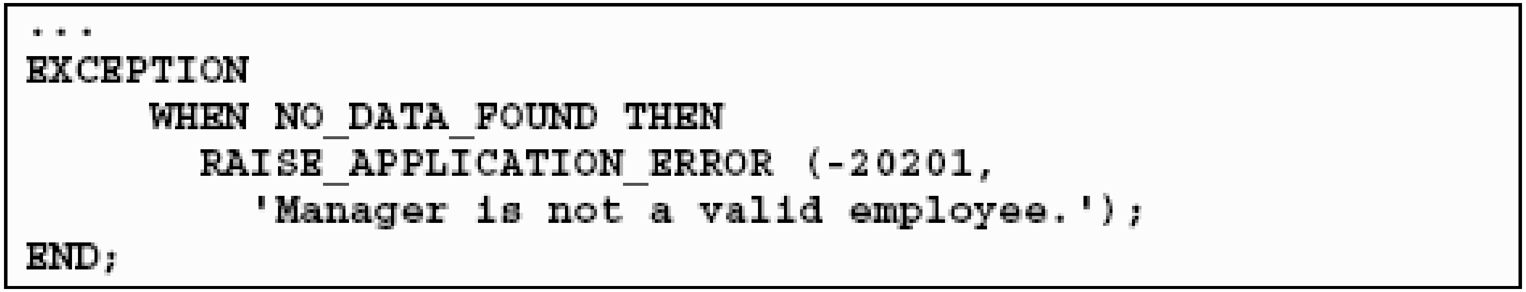
Example

Executable section:



```
BEGIN
...
  DELETE FROM employees
     WHERE  manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
     RAISE_APPLICATION_ERROR(-20202,
       'This is not a valid manager');
  END IF;
   ...
```

Exception section:



```
...
EXCEPTION
     WHEN NO_DATA_FOUND THEN
       RAISE_APPLICATION_ERROR (-20201,
         'Manager is not a valid employee.');
END;
```

Procedures (1)

- Behave very much like procedures in other programming language
- A procedure is introduced by the keywords CREATE PROCEDURE followed by the procedure name and its parameters.
- CREATE may be followed by OR REPLACE.
  - If the procedure is already created, we will not get an error
  - If the previous definition is a different procedure with the same name, the old procedure will be lost.

Procedures(2)

- Can have any number of parameters, each followed by a *mode* and a type. The possible modes are:
  - IN (read-only), OUT (write-only), and INOUT (read+write).
  - The type specifier in a parameter declaration must be unconstrained
    - CHAR(10) and VARCHAR(20) are illegal; CHAR or VARCHAR should be used instead. The actual length of a parameter depends on the corresponding argument that is passed in when the procedure is invoked.
- Following the arguments is the keyword AS (IS is a synonym). Then comes the body, which is essentially a PL/SQL block.
- The DECLARE section should not start with the keyword DECLARE. Rather, following AS we have:

```
... AS <local_var_declarations>
 BEGIN
     <procedure_body>
 END; /
```

Example

Addtuple1: given an integer i, inserts the tuple (i, 'xxx') into the following example relation:

```
CREATE TABLE T2 (a INTEGER,      b CHAR(10) );

CREATE PROCEDURE addtuple1(i IN NUMBER) AS
    BEGIN
       INSERT INTO T2 VALUES(i, 'xxx');
    END;
.
run;
```

Executing procedures

- The “run” at the end runs the statement that *creates* the procedure; it does not execute the procedure.
- To execute the procedure, use another PL/SQL statement, in which the procedure is invoked as an executable statement.

Ex:

```
BEGIN
    addtuple1(99);
END; /
```

- Or invoke the “execute” command

Ex:

```
EXECUTE addtuple1(99);
```

Another example

```
CREATE PROCEDURE addtuple2( x T2.a%TYPE, y T2.b%TYPE) AS
BEGIN
     INSERT INTO T2(a, b) VALUES(x, y);
END;
.
run;
/* to add a tuple (10, 'abc') to T2 */
BEGIN
  addtuple2(10, 'abc');
END; . run;
```

OUT and INOUT parameters

- Assigning values to parameters declared as OUT or INOUT causes the corresponding input arguments to be written.
- Because of this, the input argument for an OUT or INOUT parameter should be something with an "lvalue”.
- A constant or a literal argument should not be passed in for an OUT/INOUT parameter.

Example

```
CREATE TABLE T3 (a INTEGER, b INTEGER );

CREATE PROCEDURE addtuple3(a NUMBER, b OUT
  NUMBER) AS
 BEGIN
   b := 4;
  INSERT INTO T3 VALUES(a, b);
 END;
 . run;

DECLARE
 v NUMBER;
BEGIN
 addtuple3(10, v);
END; . run;
```

Procedures and Functions

- We can also write functions instead of procedures. In a function declaration, we follow the parameter list by RETURN and the type of the return value:

```
CREATE FUNCTION <func_name>(<param_list>) RETURN
  <return_type> AS ...
```

- In the body of the function definition, "RETURN <expression>;" exits from the function and returns the value of <expression>.
- To find out what procedures and functions you have created, use the following SQL query:

```
select object_type, object_name from user_objects
  where object_type = 'PROCEDURE' or object_type
  = 'FUNCTION';
```

- To drop a stored procedure/function:

```
drop procedure <procedure_name>;
drop function <function_name>;
```

Discovering errors

- PL/SQL does not always tell you about compilation errors. Instead, it gives you a cryptic message such as "procedure created with compilation errors". If you don't see what is wrong, try issuing the command:

```
show errors procedure
  <procedure_name>;
```

- Alternatively, you can type:

`SHO ERR` (short for SHOW ERRORS)

to see the most recent compilation error.

References

- Notes from Univ. Of Stanford: “Using Oracle PL/SQL”,

  http://www-db.stanford.edu/~ullman/fcdb/oracle/or-plsql.html
- “Oracle 9i: Program with PL/SQL”, Instructor Guide, Volume 1, ORACLE