PL/SQL

Raise Statement

- The **RAISE** statement is used to explicitly raise an exception within a PL/SQL block.
- It immediately stops normal execution of a PL/SQL block or subprogram and transfers control to an exception handler.
- It can be used to raise both system defined and user defined exceptions.
- If an exception is raised and PL/SQL cannot find a handler for it in the current block, the exception then propagates to successive enclosing blocks, until a handler is found or there are no more blocks to propagate to.
- If no handler is found, PL/SQL returns an **unhandled exception error** to the host environment.

RAISE [EXCEPTION NAME]

The PL/SQL block below selects an employee corresponding to a given employee IDd. If no employee record is found it raises the NO_DATA_FOUND exception and displays a message. Note that NO_DATA_FOUND is a system defined exception.

```
DECLARE
L_EMP VARCHAR2(1000);
CURSOR C IS
SELECT ENAME
FROM EMPLOYEE
WHERE EMPNO = 300;
BEGIN
OPEN C;
FETCH C INTO L_EMP;
CLOSE C;
IF L_EMP IS NULL THEN
RAISE NO_DATA_FOUND;
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('No Employee exists for this employee ID.');
END;
No Employee exists for this employee ID.
PL/SQL procedure successfully completed.
```

PL/SQL Tables and Records:

- Objects of type TABLE are called PL/SQL tables, which are modeled as (but not the same as) database tables.
- For example, a PL/SQL table of employee names is modeled as a database table with two columns, which store a primary key and character data, respectively. Although you cannot use SQL statements to manipulate a PL/SQL table, its primary key gives you array-like access to rows. Think of the key and rows as the index and elements of a one-dimensional array.
- Like an array, a PL/SQL table is an ordered collection of elements of the same type. Each element has a unique index number that determines its position in the ordered collection.
- However, PL/SQL tables differ from arrays in two important ways. First, arrays have fixed lower and upper bounds, but PL/SQL tables are unbounded. So, the size of a PL/SQL table can increase dynamically.
- Second, arrays require consecutive index numbers, but PL/SQL tables do not. This characteristic, called *sparsity*, allows the use of meaningful index numbers. For example, you can use a series of employee numbers (such as 7369, 7499, 7521, 7566, ...) to index a PL/SQL table of employee names.

Declaring PL/SQL Tables

- To create PL/SQL tables, you take two steps. First, you define a TABLE type, then declare PL/SQL tables of that type. You can define TABLE types in the declarative part of any block, subprogram, or package using the syntax
- TYPE table_type_name IS TABLE OF datatype [NOT NULL] INDEX BY BINARY_INTEGER; where *table_type_name* is a type specifier used in subsequent declarations of PL/SQL tables.
- The INDEX BY clause must specify datatype BINARY_INTEGER, which has a magnitude range of -2147483647 .. 2147483647. If the element type is a record type, every field in the record must have a scalar datatype such as CHAR, DATE, or NUMBER.
- To specify the element type, you can use %TYPE to provide the datatype of a variable or database column. In the following example, you define a TABLE type based on the *ename* column:

  DECLARE TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;

A RECORD type to specify the element type:

```
DECLARE TYPE TimeRecTyp IS RECORD (
hour SMALLINT := 0,
minute SMALLINT := 0,
second SMALLINT := 0);
TYPE TimeTabTyp IS TABLE OF TimeRecTyp
INDEX BY BINARY_INTEGER;
```

Declaring PL/SQL Tables

- Once you define a TABLE type, you can declare PL/SQL tables of that type, as the following examples show:

```
DECLARE TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
INDEX BY BINARY_INTEGER;
TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE INDEX
BY BINARY_INTEGER;
sal_tab SalTabTyp; -- declare PL/SQL table
emp_tab EmpTabTyp; -- declare another PL/SQL table
```

Referring PL/SQL Tables

- To reference elements in a PL/SQL table, you specify an index number using the syntax

  plsql_table_name(index)

  where *index* is an expression that yields a BINARY_INTEGER value or a value implicitly convertible to that datatype. In the following example, you reference an element in the PL/SQL table *hiredate_tab*:

  hiredate_tab(i + j - 1) ...

  As the example below shows, the index number can be negative.

  hiredate_tab(-5) ...

  The following example shows that you can reference the elements of a PL/SQL table in subprogram calls:

  raise_salary(empno_tab(i), amount); -- call subprogram

Inserting and Fetching rows using PL/SQL table

- You must use a loop to insert values from a PL/SQL table into a database column. For example, given the declarations

  CREATE PACKAGE emp_defs AS TYPE EmpnoTabTyp IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;

  TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;

  empno_tab EmpnoTabTyp;

  ename_tab EnameTabTyp; ...

  END emp_defs;

- you might use the following standalone procedure to insert values from the PL/SQL tables *empno_tab* and *ename_tab* into the database table *emp*:

  CREATE PROCEDURE insert_emp_ids (rows IN BINARY_INTEGER,

  empno_tab IN EmpnoTabTyp, ename_tab IN EnameTabTyp) AS BEGIN FOR i IN 1..rows LOOP INSERT INTO emp (empno, ename) VALUES (empno_tab(i), ename_tab(i));

  END LOOP;

  END;

Deleting Rows

- This attribute has three forms. DELETE removes all elements from a PL/SQL table. DELETE(*n*) removes the *n*th element. If *n* is null, DELETE(*n*) does nothing. DELETE(*m*, *n*) removes all elements in the range *m* .. *n*.
- If *m* is larger than *n* or if *m* or *n* is null, DELETE(*m*, *n*) does nothing.DELETE lets you free the resources held by a PL/SQL table. DELETE(*n*) and DELETE(*m*, *n*) let you prune a PL/SQL table. Consider the following examples:

```
ename_tab.DELETE(3); -- delete element 3

ename_tab.DELETE(5, 5); -- delete element 5

ename_tab.DELETE(20, 30); -- delete elements 20 through 30

ename_tab.DELETE(-15, 0); -- delete elements -15 through 0
ename_tab.DELETE; -- delete entire PL/SQL table
```

Declaration of Records

- Records must be declared in two steps. First, you define a RECORD type, then declare user-defined records of that type. You can define RECORD types in the declarative part of any block, subprogram, or package using the syntax

  TYPE record_type_name IS RECORD (field[, field]...);

  where *record_type_name* is a type specifier used in subsequent declarations of records and *field* stands for the following syntax:

  field_name datatype [[NOT NULL] {:= | DEFAULT} expr]

- You can use the attributes %TYPE and %ROWTYPE to specify field types. In the following example, you define a RECORD type named *DeptRecTyp*:

  DECLARE TYPE DeptRecTyp IS RECORD

  (deptno NUMBER(2), dname dept.dname%TYPE, loc dept.loc%TYPE);

Functions

```
DECLARE TYPE EmpRecTyp IS RECORD ( emp_id NUMBER(6), salary NUMBER(8,2));

CURSOR desc_salary RETURN EmpRecTyp IS SELECT employee_id, salary FROM employees ORDER BY salary DESC;

emp_rec EmpRecTyp; FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS BEGIN OPEN desc_salary;

FOR i IN 1..n LOOP FETCH desc_salary INTO emp_rec; END LOOP;

CLOSE desc_salary;

RETURN emp_rec;

END nth_highest_salary;

BEGIN NULL;

END;

/
```

Parameters in Procedure and Functions

- In PL/SQL, we can pass parameters to procedures and functions in three ways.

  1) IN type parameter: These types of parameters are used to send values to stored procedures.
  2) OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
  3) IN OUT parameter: These types of parameters are used to send values and get values from stored procedures.

  NOTE: If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

General syntax to pass a IN parameter is

CREATE [OR REPLACE] PROCEDURE procedure_name (param_name1 IN datatype, param_name12 IN datatype ...)

General syntax to create an OUT parameter is

CREATE [OR REPLACE] PROCEDURE proc2 (param_name OUT datatype)

General syntax to create an IN OUT parameter is

CREATE [OR REPLACE] PROCEDURE proc3 (param_name IN OUT datatype)

Purity Functions

- To be callable from SQL statements, a stored function must obey certain "purity" rules, which are meant to control side effects.
- When called from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.
- When called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.
- When called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). Also, it cannot execute DDL statements (such as CREATE) because they are followed by an automatic commit.

Packages

- A package is a schema object that groups logically related PL/SQL types, variables, and subprograms.
- Packages usually have two parts, a specification (spec) and a body; sometimes the body is unnecessary.
- The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
- The body defines the queries for the cursors and the code for the subprograms.

Package Specification

- Get and Set methods for the package variables, if you want to avoid letting other procedures read and write them directly.
- Cursor declarations with the text of SQL queries. Reusing exactly the same query text in multiple locations is faster than retyping the same query each time with slight differences. It is also easier to maintain if you need to change a query that is used in many places.
- Declarations for exceptions. Typically, you need to be able to reference these from different procedures, so that you can handle exceptions within called subprograms.
- Declarations for procedures and functions that call each other. You do not need to worry about compilation order for packaged procedures and functions, making them more convenient than standalone stored procedures and functions when they call back and forth to each other.
- Declarations for overloaded procedures and functions. You can create multiple variations of a procedure or function, using the same names but different sets of parameters.
- Variables that you want to remain available between procedure calls in the same session. You can treat variables in a package like global variables.
- Type declarations for PL/SQL collection types. To pass a collection as a parameter between stored procedures or functions, you must declare the type in a package so that both the calling and called subprogram can refer to it.

Advantages of PL/SQL Packages

- **Modularity:** Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.
- **Easier Application Design:** When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.
- **Information Hiding:** With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the implementation of the private subprogram so that only the package (not your application) is affected if the implementation changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.
- **Added Functionality:** Packaged public variables and cursors persist for the duration of a session. They can be shared by all subprograms that execute in the environment. They let you maintain data across transactions without storing it in the database.
- **Better Performance:** When you call a packaged subprogram for the first time, the whole package is loaded into memory. Later calls to related subprograms in the package require no disk I/O.

Private Versus Public Package Objects

- For objects that are declared inside the package body, you are restricted to use within that package.
- Therefore, PL/SQL code outside the package cannot reference any of the variables that were privately declared within the package.
- Any items declared inside the package specification are visible outside the package.
- These objects declared in the package specification are called public.
- If the variable, constant, or cursor was declared in a package specification or body, their values persist for the duration of the user's session.
- The values are lost when the current user's session terminates or the package is recompiled.

Cursors in Packages

- When you use global cursor in package, you should determine where cursor declaration in your code. Some developers declare cursor in package specification and others in package body.

  CREATE PACKAGE pack_name AS CURSOR c1 is select * from emp;

  ... END pack_name;

  CREATE PACKAGE BODY pack_name AS CURSOR c1 is select * from emp;

  ... END pack_name;