PL/SQL

PL/SQL stands for Procedural Language/SQL.PL/SQL extends SQL by adding control Structures found in other procedural language.PL/SQL combines the flexibility of SQL with Powerful feature of 3rd generation Language. The procedural construct and database access Are present in PL/SQL.PL/SQL can be used in both in database in Oracle Server and in Client side application development tools.

Advantages of PL/SQL

Support for SQL, support for object-oriented programming,, better performance, portability, higher productivity, Integration with Oracle

a] Supports the declaration and manipulation of object types and collections.
b] Allows the calling of external functions and procedures.
c] Contains new libraries of built in packages.
d] with PL/SQL , an multiple sql statements can be processed in a single command line statement.

PL/SQL Datatypes

Scalar Types
BINARY_INTEGER ,DEC,DECIMAL,DOUBLE ,,PRECISION,FLOAT,INT,INTEGER,NATURAL,
NATURALN,NUMBER, NUMERIC, PLS_INTEGER,POSITIVE,POSITIVEN,REAL,SIGNTYPE,
SMALLINT,CHAR,CHARACTER,LONG,LONG RAW,NCHAR,NVARCHAR2,RAW,ROWID,STRING,
VARCHAR,VARCHAR2,

Composite Types
TABLE, VARRAY, RECORD

LOB Types
BFILE, BLOB, CLOB, NCLOB

Reference Types
REF CURSOR

BOOLEAN, DATE

DBMS_OUTPUT.PUT_LINE:

It is a pre-defined package that prints the message inside the parenthesis

ANONYMOUS PL/SQL BLOCK.

The text of an Oracle Forms trigger is an anonymous PL/SQL block. It consists of three sections:

- A declaration of variables, constants, cursors and exceptions which is optional.
- A section of executable statements.
- A section of exception handlers, which is optional.

ATTRIBUTES

Allow us to refer to data types and objects from the database.PL/SQL variables and Constants can have attributes. The main advantage of using Attributes is even if you Change the data definition, you don't need to change in the application.

%TYPE

It is used when declaring variables that refer to the database columns.
Using %TYPE to declare variable has two advantages. First, you need not know the exact datatype of variable. Second, if the database definition of variable changes, the datatype of variable changes accordingly at run time.

%ROWTYPE

The %ROWTYPE attribute provides a record type that represents a row in a table (or view). The record can store an entire row of data selected from the table or fetched from a cursor or strongly typed cursor variable.

EXCEPTION

An Exception is raised when an error occurs. In case of an error then normal execution stops and the control is immediately transferred to the exception handling part of the PL/SQL Block.

Exceptions are designed for runtime handling, rather than compile time handling. Exceptions improve readability by letting you isolate error-handling routines.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the runtime system. User-defined exceptions must be raised explicitly by RAISE statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called exception handlers. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

Exception Types

1. Predefined Exceptions

An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.

2. User – Defined exceptions

User – defined exception must be defined and explicitly raised by the user

EXCEPTION_INIT

A named exception can be associated with a particular oracle error. This can be used to trap the error specifically.

PRAGMA EXCEPTION_INIT(exception name, Oracle_error_number);

The pragma EXCEPTION_INIT associates an exception name with an Oracle, error number. That allows you to refer to any internal exception by name and to write a specific handler

RAISE_APPLICATION_ERROR

The procedure raise_application_error lets you issue user-defined error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To call raise_application_error, you use the syntax

raise_application_error(error_number, message[, {TRUE | FALSE}]);
where error_number is a negative integer in the range -20000 .. -20999 and message is a character string up to 2048 bytes long.

Exception	Raised when ...
ROWTYPE_MISMATCH	the host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. For example, when you pass an open host cursor variable to a stored subprogram, the return types of the actual and formal parameters must be compatible.
STORAGE_ERROR	PL/SQL runs out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	you reference a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	you reference a nested table or varray element using an index number that is outside the legal range (-1 for example).
TIMEOUT_ON_RESOURCE	a timeout occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	a SELECT INTO statement returns more than one row.
VALUE_ERROR	an arithmetic, conversion, truncation, or size-constraint error occurs. For example, when you select a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string to a number fails. In SQL statements, INVALID_NUMBER is raised.
ZERO_DIVIDE	you try to divide a number by zero.
NOT_LOGGED_ON	your PL/SQL program issues a database call without being connected to Oracle.
PROGRAM_ERROR	PL/SQL has an internal problem.

Exception	Raised when ...
ACCESS_INTO_NULL	you try to assign values to the attributes of an uninitialized (atomically null) object.
COLLECTION_IS_NULL	you you try to apply collection methods other than EXISTS to an uninitialized (atomically null) nested table or varray, or you try to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	you try to open an already open cursor. You must close a cursor before you can reopen it. A cursor FOR loop automatically opens the cursor to which it refers. So, you cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	you try to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	you try an illegal cursor operation such as closing an unopened cursor.
INVALID_NUMBER	in a SQL statement, the convertion of character string to a number fails because the character string does not represent a valid number. In procedural statements, VALUE_ERROR is raised.
LOGIN_DENIED	you try logging on to Oracle with an invalid username and/or password.
NO_DATA_FOUND	a SELECT INTO statement returns no rows, or you reference a deleted element in a nested table, or you reference an uninitialized element in an index-by table. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised. SQL group functions such as AVG and SUM *always* return a value or a null. So, a SELECT INTO statement that calls a group function will never raise NO_DATA_FOUND.

Using SQLCODE and SQLERRM

For internal exceptions, SQLCODE returns the number of the Oracle error. The number that SQLCODE returns is negative unless the Oracle error is no data found, in which case SQLCODE returns +100. SQLERRM returns the corresponding error message. The message begins with the Oracle error code.

Unhandled Exceptions

PL/SQL returns an unhandled exception error to the host environment, which determines the outcome.

When Others

It is used when all exception are to be trapped.

CURSORS

Oracle allocates an area of memory known as context area for the processing of SQL statements. The pointer that points to the context area is a cursor.

Merits

1] Allowing to position at specific rows of the result set.
2] Returning one row or block of rows from the current position in the result set.
3] Supporting data modification to the rows at the current position in the result set.

TYPES

1] STATIC CURSOR

SQL statement is determined at design time.

A] EXPLICIT CURSOR

Multiple row SELECT statement is called as an explicit cursor.

To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use an explicit cursor, which names the work area.

Usage - If the SELECT statement returns more that one row then explicit cursor should be used.

Steps

- Declare a cursor
- Open a cursor
- Fetch data from the cursor
- Close the cursor

EXPLICIT CURSOR ATTRIBUTES

- %FOUND (Returns true if the cursor has a value)
- %NOTFOUND (Returns true if the cursor does not contain any value)
- %ROWCOUNT (Returns the number of rows selected by the cursor)
- %ISOPEN (Returns the cursor is opened or not)

CURSOR FOR LOOP

The CURSOR FOR LOOP lets you implicitly OPEN a cursor, FETCH each row returned by the query associated with the cursor and CLOSE the cursor when all rows have been processed.

SYNTAX

```
FOR <RECORD NAME> IN <CURSOR NAME> LOOP
      STATEMENTS
END LOOP;
```

To refer an element of the record use <***record name. Column name***>

Parameterized Cursor

A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be IN parameters. Therefore, they cannot return values to actual parameters. Also, you cannot impose the constraint NOT NULL on a cursor parameter.

The values of cursor parameters are used by the associated query when the cursor is opened.

B .IMPLICIT CURSOR

An IMPLICIT cursor is associated with any SQL DML statement that does not have a explicit cursor associated with it.

This includes:

- All INSERT statements
- All UPDATE statements
- All DELETE statements
- All SELECT .. INTO statements

IMPLICIT CURSOR ATTRIBUTES
" SQL%FOUND (Returns true if the DML operation is valid)
" SQL%NOTFOUND (Returns true if the DML operation is invalid)
" SQL%ROWCOUNT (Returns the no. of rows affected by the DML operation)

2] DYNAMIC CURSOR

Dynamic Cursor can be used along with DBMS_SQL package .A SQL statement is dynamic, if it is constructed at run time and then executed.

3] REF CURSOR

Declaring a cursor variable creates a pointer, not an item. In PL/SQL, a pointer has datatype REF X, where REF is short for REFERENCE and X stands for a class of objects. Therefore, a cursor variable has datatype REF CURSOR.

To execute a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use an explicit cursor, which names the work area. Or, you can use a cursor variable, which points to the work area.
Mainly, you use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle Server can all refer to the same work area.

a] **Strong Cursor** - Whose return type specified.
b] **Week Cursor** - Whose return type not specified.

Dref
It is the 'deference operator.Like VALUE,it return the value of an object,Unlike value.
Dref's input is a REF to an column in a table and you want reterive the target instead of the pointer,you DREF.

ORACLE 9I

Advanced Explicit Cursor Concepts

1. FOR UPDATE WAIT Clause

- Lock the rows before the update or delete.
- Use explicit locking to deny access for the duration of a transaction.

Syntax

Cursor eipc1 is select * from emp for update [of column_reference] [NOWAIT]

Column Reference — It is column in the table against which the query is performed [A list of column may also be used]

NOWAIT — Returns an Oracle Error if the rows are locked by another session.

- The SELECT... FOR UPDATE statement has been modified to allow the user to specify how long the command should wait if the rows being selected are locked.

- If NOWAIT is specified, then an error is returned immediately if the lock cannot be obtained.

Example of Using FOR UPDATE WAIT Clause

1. SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = 10 FOR UPDATE WAIT 20;

2.
```
DECLARE
        CURSOR EMP_CURSOR IS
        SELECT EMPNO, ENAME, DNAME FROM EMP,DEPT WHERE
        EMP.DEPTNO=DEPT.DEPTNO AND EMP.DEPTNONO=80
        FOR UPDATE OF SALARY NOWAIT;
```

[*Retrieve the Employees who work in department 80 and update their Salary]*

2. The WHERE CURRENT OF Clause

- Use cursors to update or delete the cursor row.
- Include the FOR UPDATE clause in the cursor query to lock the row first.
- Use the WHERE CURRENT OF clause to refer the current row from an explicit cursor.

Syntax

```
WHERE CURRENT OF < cursor_name >
```

Cursor_Name -It is the name of a declared cursor. [The cursor have been declared with the FOR UPDATE clause]

Example of Using FOR WHERE CURRENT OF Clause

1.
```
DECLARE
        CURSOR EMP_CURSOR IS SELECT EMPNO, ENAME, DNAME FROM E.EMP,D.DEPT WHERE
        EMP.DEPTNO=DEPT.DEPTNO AND EMP.DEPTNONO=80 FOR UPDATE OF SALARY NOWAIT;
BEGIN
        FOR I IN EMP_CURSOR;
        LOOP
                IF I.SAL<5000 THEN

                 UPDATE EMP SET SALARY=I.SAL*1.10 WHERE CURRENT OF EMP_CURSOR;

                END IF;
        END LOOP;
END;
```

[*The Example loops through each employee in department 80 , and checks whether the salary is less than 5000.If salary is less than , the salary is raised by 10%. The where current of clause in the UPDATE statement refers to the currently fetched records.]*

3. CURSORS WITH SUB QUERIES

Sub queries are often used in the WHERE clause of select statement. It can be used to FROM clause, creating a temporary data source for the query.

```
DECLARE
      bonus REAL;
BEGIN
      FOR emp_rec IN (SELECT empno, sal, comm FROM emp)
      LOOP
            bonus := (emp_rec.sal * 0.05) + (emp_rec.comm * 0.25);
            INSERT INTO bonuses VALUES (emp_rec.empno, bonus);

      END LOOP;
      COMMIT;
END;
```

PROCEDURES

Procedure is a subprogram that contains set of SQL and PL/SQL statements.

Merits

Reusability - Subprograms once executed can be used in any number of applications
Maintainability - Subprogram can simplify maintenance, subprogram will be affected only its definition changes

Parameter

IN - The IN parameter is used to pass values to a subprogram when it is invoked
OUT - The OUT parameter is used to return values to the caller of a subprogram
IN OUT The IN OUT parameter is used to pass initial values to the subprogram when invoked and also it returns updated values to the caller.

Using the NOCOPY Compiler Hint

Suppose a subprogram declares an IN parameter, an OUT parameter, and an IN OUT parameter. When you call the subprogram, the IN parameter is passed by reference. That is, a pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both parameters reference the same memory location, which holds the value of the actual parameter.

By default, the OUT and IN OUT parameters are passed by value. That is, the value of the IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

When the parameters hold large data structures such as collections, records, and instances of object types, all this copying slows down execution and uses up memory. To prevent that, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. In the following example, you ask the compiler to pass IN OUT parameter my_staff by reference instead of by value:

```
DECLARE
TYPE Staff IS VARRAY(200) OF Employee;
PROCEDURE reorganize (my_staff IN OUT NOCOPY Staff) IS ...
```

Remember, NOCOPY is a hint, not a directive. So, the compiler might pass my_staff by value despite your request. Usually, however, NOCOPY succeeds. So, it can benefit any PL/SQL application that passes around large data structures.

In the example below, 25000 records are loaded into a local nested table, which is passed to two local procedures that do nothing but execute NULL statements. However, a call to one procedure takes 21 seconds because of all the copying. With NOCOPY, a call to the other procedure takes much less than 1 second.

```
SQL> SET SERVEROUTPUT ON
SQL> GET test.sql
1 DECLARE
2     TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE;
3     emp_tab EmpTabTyp := EmpTabTyp(NULL); -- initialize
4     t1 NUMBER(5);
5     t2 NUMBER(5);
6     t3 NUMBER(5);
7 PROCEDURE get_time (t OUT NUMBER) IS
8     BEGIN
            SELECT TO_CHAR(SYSDATE,'SSSSS') INTO t FROM dual;
      END;
9 PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
10    BEGIN
            NULL;
      END;

11 PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
```

```
12    BEGIN
            NULL;
      END;
13 BEGIN
14    SELECT * INTO emp_tab(1) FROM emp WHERE empno = 7788;
15    emp_tab.EXTEND(24999, 1); -- copy element 1 into 2..25000
16    get_time(t1);
17    do_nothing1(emp_tab); -- pass IN OUT parameter
18    get_time(t2);
19    do_nothing2(emp_tab); -- pass IN OUT NOCOPY parameter
20    get_time(t3);
21    DBMS_OUTPUT.PUT_LINE('Call Duration (secs)');
22    DBMS_OUTPUT.PUT_LINE('--------------------');
23    DBMS_OUTPUT.PUT_LINE('Just IN OUT: ' || TO_CHAR(t2 - t1));
24    DBMS_OUTPUT.PUT_LINE('With NOCOPY: ' || TO_CHAR(t3 - t2));
25* END;
SQL> /

Call Duration (secs)
--------------------
Just IN OUT: 21
With NOCOPY: 0
```

Autonomous Transactions

A transaction is a series of SQL statements that does a logical unit of work. Often, one transaction starts another. In some applications, a transaction must operate outside the scope of the transaction that started it. This can happen, for example, when a transaction calls out to a data cartridge.

An *autonomous transaction* is an independent transaction started by another transaction, the *main transaction*. Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction.

Advantages of Autonomous Transactions

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. So, you can log events, increment retry counters, and so on, even if the main transaction rolls back.

More important, autonomous transactions help you build modular, reusable software components. For example, stored procedures can start and finish autonomous transactions on their own. A calling application need not know about a procedure's autonomous operations, and the procedure need not know about the application's transaction context. That makes autonomous transactions less Error-prone than regular transactions and easier to use.

Furthermore, autonomous transactions have all the functionality of regular transactions. They allow parallel queries, distributed processing, and all the transaction control statements including SET TRANSACTION.

VIEW SOURCE CODE FOR PROCEDURES

```
SELECT TEXT FROM USER_SOURCE WHERE NAME='&P1' AND TYPE='PROCEDURE'
```

FUNCTIONS

Function is a subprogram that computes and returns a single value.
A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions have a RETURN clause.

```
FUNCTION NAME [(PARAMETER[, PARAMETER, ...])] RETURN DATATYPE IS
[LOCAL DECLARATIONS]
BEGIN
      EXECUTABLE STATEMENTS
[EXCEPTION
      EXCEPTION HANDLERS]
```

A function has two parts: the specification and the body. The function specification begins with the keyword FUNCTION and ends with the RETURN clause, which specifies the datatype of the result value. Parameter declarations are optional. Functions that take no parameters are written without parentheses. The function body begins with the keyword IS and ends with the keyword END followed by an optional function name.

Using the RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. (Do not confuse the RETURN statement with the RETURN clause in a function spec, which specifies the datatype of the return value.)

A subprogram can contain several RETURN statements, none of which need be the last lexical statement. Executing any of them completes the subprogram immediately. However, to have multiple exit points in a subprogram is a poor programming practice.

In procedures, a RETURN statement cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.

However, in functions, a RETURN statement *must* contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, which acts like a variable of the type specified in the RETURN clause. Observe how the function balance returns the balance of a specified bank account:

COMPARING PROCEDURES AND FUNCTIONS

Procedure	Function
Execute as a PL/SQL statement	Invoke as part of an expression
No RETURN datatype	Must contain a RETURN datatype
Can return none, one or many values	Must return a single value
Can not use in SQL Statement	Can Use in SQL Statements

Benefits of Stored Procedures and Functions

In addition to modularizing application development, stored procedures and functions have the following benefits:

- **Improved performance**
  - Avoid reparsing for multiple users by exploiting the shared SQL area
  - Avoid PL/SQL parsing at run-time by parsing at compile time
  - Reduce the number of calls to the database and decrease network traffic by bundling commands
- **Improved maintenance.**
  - Modify routines online without interfering with other users
  - Modify one routine to affect multiple applications
  - Modify one routine to eliminate duplicate testing
- **Improved data security and integrity**

- Control indirect access to database objects from non privileged users with security privileges
- Ensure that related actions are performed together, or not at all, by funneling activity for related tables through a single path

LIST ALL PROCEDURES AND FUNCTIONS

```
SELECT OBJECT_NAME, OBJECT_TYPE FROM USER_OBJECTS WHERE OBJECT_TYPE IN ('PROCEDURE', 'FUNCTION') ORDER BY OBJECT_NAME
```

LIST THE CODE OF PROCEDURES AND FUNCTIONS

```
SELECT TEXT FROM USER_SOURCE WHERE NAME = 'QUERY_EMP' ORDER BY LINE;
```

PACKAGE

A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages usually have two parts, a specification and a body,

Merits

Simplify security	No need to issue GRANT for every procedure in a package
Limit recompilation	Change the body of the procedure or function without changing specification
Performance	First call to package loads whole package into memory
Information Hiding	With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible).
Hiding	Implementation details from users, you protect the integrity of the package.

Parts of Package

A.PACKAGE SPECIFICATION

The package specification contains public declarations. The scope of these declarations is local to your database schema and global to the package. So, the declared items are accessible from your application and from anywhere in the package.

B.PACKAGE BODY

The package body implements the package specification. That is, the package body contains the definition of every cursor and subprogram declared in the package specification. Keep in mind that subprograms defined in a package body are accessible outside the package only if their specifications also appear in the package specification.

PACKAGE OVERLOADING

PL/SQL allows two or more packaged subprograms to have the same name. This option is useful when you want a subprogram to accept parameters that have different datatypes.

PRIVATE VERSUS PUBLIC ITEMS

PRIVATE
The package body can also contain private declarations, which define types and items necessary for the internal workings of the package. The scope of these declarations is local to the package body. Therefore, the declared types and items are inaccessible except from within the package body. Unlike a package spec, the declarative part of a package body can contain subprogram bodies.

PUBLIC
Such items are termed *public*. When you must maintain items throughout a session or across transactions, place them in the declarative part of the package body.

Advantages of Packages

Packages offer several advantages: modularity, easier application design, information hiding, added functionality, and better performance.

- **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. So, they can be shared by all subprograms that execute in the environment. Also, they allow you to maintain data across transactions without having to store it in the database.

- **Better Performance**
  When you call a packaged subprogram for the first time, the whole package is loaded into memory. So, later calls to related subprograms in the package require no disk I/O. Also, packages stop cascading dependencies and thereby avoid unnecessary recompiling. For example, if you change the implementation of a packaged function, Oracle need not recompile the calling subprograms because they do not depend on the package body.

ORACLE SUPPLIED PACKAGES

- Provided with the Oracle Server
- Extend the functionality of the database
- Allow access to certain SQL features normally restricted for PL/SQL

Use Serially Reusable Packages

To help you manage the use of memory, PL/SQL provides the pragma SERIALLY_ REUSABLE, which lets you mark some packages as *serially reusable.* You can so mark a package if its state is needed only for the duration of one call to the server (for example, an OCI call to the server or a server-to-server RPC).

The global memory for such packages is pooled in the System Global Area (SGA), not allocated to individual users in the User Global Area (UGA). That way, the package work area can be reused. When the call to the server ends, the memory is returned to the pool. Each time the package is reused, its public variables are initialized to their default values or to NULL.

The maximum number of work areas needed for a package is the number of concurrent users of that package, which is usually much smaller than the number of logged-on users. The increased use of SGA memory is more than offset by the decreased use of UGA memory. Also, Oracle ages-out work areas not in use if it needs to reclaim SGA memory.

For bodiless packages, you code the pragma in the package spec using the following syntax:

```
      PRAGMA SERIALLY_REUSABLE;
```

For packages with a body, you must code the pragma in the spec and body. You cannot code the pragma only in the body. The following example shows how a public variable in a serially reusable package behaves across call boundaries:

```
CREATE PACKAGE pkg1 IS
      PRAGMA SERIALLY_REUSABLE;
      num NUMBER := 0;
      PROCEDURE init_pkg_state(n NUMBER);
      PROCEDURE print_pkg_state;
END pkg1;
```

DATABASE TRIGGERS

A Trigger defines an action the database should take when some database related event occurs. Triggers may be used to supplement declarative referential integrity, to enforce complex business rules.

A database trigger is a stored subprogram associated with a table. You can have Oracle automatically fire the trigger before or after an INSERT, UPDATE, or DELETE statement affects the table.

Triggers are executed when a specific data manipulation command are performed on specific tables

ROW LEVEL TRIGGERS

Row Level triggers execute once for each row in a transaction. Row level triggers are create using FOR EACH ROW clause in the create trigger command.

STATEMENT LEVEL TRIGGERS

Statement Level triggers execute once for each transaction. For example if you insert 100 rows in a single transaction then statement level trigger will be executed once.

BEFORE and AFTER Triggers

Since triggers occur because of events, they may be set to occur immediately before or after those events.

The following table shows the number of triggers that you can have for a table. The number of triggers you can have a for a table is 14 triggers.

The following table shows the number of triggers that you can have for a table. The number of triggers you can have a for a table is 14 triggers.

	ROW LEVEL TRIGGER	STATEMENT LEVEL TRIGGER
BEFORE	INSERT UPDATE DELETE	INSERT UPDATE DELETE
INSTEAD OF	INSTEAD OF ROW	
AFTER	INSERT UPDATE DELETE	INSERT UPDATE DELETE

ADVANTAGES OF TRIGGERS

Feature	**Enhancement**
Security	The Oracle Server allows table access to users or roles. Triggers allow table access according to data values.
Auditing	The Oracle Server tracks data operations on tables. Triggers track values for data operations on tables.
Data integrity	The Oracle Server enforces integrity constraints. Triggers implement complex integrity rules.
Referential integrity	The Oracle Server enforces standard referential integrity rules. Triggers implement nonstandard functionality.
Table replication	The Oracle Server copies tables asynchronously into snapshots. Triggers copy tables

Synchronously into replicas.

Derived data The Oracle Server computes derived data values manually. Triggers compute derived data values automatically.

Event logging The Oracle Server logs events explicitly. Triggers log events transparently.

Syntax:

```
CREATE OR REPLACE TRIGGER <TRIGGER NAME> [ BEFORE | AFTER | INSTEAD OF ]
(INSERT OR UPDATE OR DELETE) ON <TABLE NAME>  REFERENCING OLD AS OLD | NEW AS NEW  FOR EACH ROW
BEGIN
END;
```

Example 1:

```
CREATE OR REPLACE TRIGGER MY_TRIG BEFORE INSERT OR UPDATE OR DELETE ON
     DETAIL FOR EACH ROW
    BEGIN
      IF LTRIM(RTRIM(TO_CHAR(SYSDATE,'DAY')))='FRIDAY' THEN
        IF INSERTING THEN
          RAISE_APPLICATION_ERROR(-20001,'INSERT IS NOT POSSIBLE');
        ELSIF UPDATING THEN
          RAISE_APPLICATION_ERROR(-20001,'UPDATE IS NOT POSSIBLE');
        ELSIF DELETING THEN
          RAISE_APPLICATION_ERROR(-20001,'DELETE IS NOT POSSIBLE');
        END IF;
      END IF;
   END;
```

Example 2:

```
CREATE OR REPLACE TRIGGER MY_TRIG AFTER INSERT ON
    ITEM FOR EACH ROW
  DECLARE
    MITEMID NUMBER;
    MQTY NUMBER;
   BEGIN
    SELECT ITEMID INTO MITEMID FROM STOCK WHERE ITEMID = :NEW.ITEMID;
       UPDATE STOCK SET QTY=QTY+:NEW.QTY WHERE ITEMID=:NEW.ITEMID;
   EXCEPTION WHEN NO_DATA_FOUND THEN
      INSERT INTO STOCK VALUES (:NEW.ITEMID, :NEW.QTY);
   END;
```

Example 3:

```
CREATE OR REPLACE TRIGGER MY_TRIG AFTER DELETE ON
    EMP FOR EACH ROW
BEGIN
      INSERT INTO EMP_BACK VALUES (:OLD.EMPNO, :OLD.ENAME, :OLD.SAL, :OLD.DEPTNO);
END;
```

Example 4:

```
CREATE OR REPLACE TRIGGER TR02 BEFORE INSERT OR UPDATE OR DELETE ON EMP100
DECLARE
      D1 VARCHAR(3);
BEGIN
      D1:=TO_CHAR(SYSDATE,'DY');

      IF D1 IN('TUE','MON') THEN
            RAISE_APPLICATION_ERROR(-20025,'TRY ON ANOTHER DAY');
      END IF;
END;
```

Example 5:

```
CREATE OR REPLACE TRIGGER TR01 AFTER DELETE ON DEPT200  FOR EACH ROW
BEGIN
      INSERT INTO DEPT1 VALUES (:OLD.DEPTNO,:OLD.DNAME,:OLD.LOC);
END;
/
SHOW ERR
```

Example 6:

```
CREATE OR REPLACE TRIGGER TR03 AFTER UPDATE ON EMP FOR EACH ROW
BEGIN
      UPDATE EMP100 SET SAL=:OLD.SAL*2  WHERE EMPNO=:OLD.EMPNO;
END;
/
SHOW ERR
```

Example 7:

```
CREATE OR REPLACE TRIGGER TR05 AFTER UPDATE ON  EMP100
DECLARE
      U1 VARCHAR2(50);
BEGIN
      SELECT USER INTO U1 FROM DUAL;
      INSERT INTO USER1 VALUES(U1,SYSDATE,'UPDATE');
END;
/
SHOW ERR
```

Example 8:

```
CREATE OR REPLACE TRIGGER TR06 AFTER DELETE ON  EMP100
DECLARE
U1 VARCHAR2(50);
BEGIN
SELECT USER INTO U1 FROM DUAL;
INSERT INTO USER1 VALUES(U1,SYSDATE,'DELETE');
END;
/
SHOW ERR
```

SYSTEM TRIGGER

LOGON and LOGOFF Trigger Example

You can create this trigger to monitor how often you log on and off, or you may want to write a report on how long you are logged on for. If you were a DBA wanting to do this, you would replace SCHEMA with DATABASE.

```
1.    CREATE OR REPLACE TRIGGER LOGON_TRIG
      AFTER logon ON SCHEMA
      BEGIN
            INSERT INTO log_trig_table 5 (user_id, log_date, action) VALUES
            (user, sysdate, 'Logging on');
      END;

2.    CREATE OR REPLACE TRIGGER LOGOFF_TRIG
      BEFORE logoff ON SCHEMA
      BEGIN
            INSERT INTO log_trig_table  (user_id, log_date, action) VALUES
            (user, sysdate, 'Logging off');
      END;
```

CALL STATEMENT

This allows you to call a stored procedure, rather than coding the PL/SQL body in the trigger itself.

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing
event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
[WHEN condition]
CALL procedure_name

CREATE TRIGGER TEST3
BEFORE INSERT ON EMP
CALL LOG_EXECUTION
```

INSTEAD OF TRIGGERS

INSTEAD OF triggers to tell ORACLE what to do instead of performing the actions that executed the trigger. An INSTEAD OF trigger can be used for a view. These triggers can be used to overcome the restrictions placed by oracle on any view which is non updateable.

Example:

```
CREATE OR REPLACE VIEW EMP_DEPT AS SELECT A.DEPTNO,B.EMPNO FROM DEPT A,EMP B
WHERE A.DEPTNO=B.DEPTNO

CREATE OR REPLACE TRIGGER TRIG1 INSTEAD OF INSERT ON EMP_DEPT
REFERENCING NEW AS N FOR EACH ROW
BEGIN
      INSERT INTO DEPT VALUES(:N.DEPTNO,'DNAME');
      INSERT INTO EMP VALUES(:N.EMPNO,'ENAME',:N.DEPTNO);
END;
```