

# C

## Overview of PL/SQL

---

### C.1 INTRODUCTION

PL/SQL is Oracle's procedural language extension to Oracle SQL. PL/SQL is a block-structured language that enables developers code procedures, functions and unnamed blocks that combine SQL with procedural statements. Related procedures, functions, cursors and variable declarations can be grouped and stored together in a package. Parameters can be passed to these blocks, enabling procedures and functions to accept and return values.

PL/SQL fully supports ANSI/ISO SQL data manipulation language commands and SQL data types. Complex data is easily handled with PL/SQL's Boolean and composite data types.

With blocks, you can group logically related declarations and statements. The declarations are local in state and cease to exist when the block completes. PL/SQL blocks can appear anywhere that SQL blocks can appear. PL/SQL is processed by its own PL/SQL engine. This engine is incorporated into the following Oracle products:

- Oracle server with procedural option
- Oracle Forms
- SQL\*Menu
- Oracle Reports
- Oracle Graphics

PL/SQL is a complete transaction processing language that provides the following advantages:

- PL/SQL supports SQL. SQL is a non-procedural language, which means that you state what you want and Oracle determines the best way to accomplish it. PL/SQL fully supports SQL data manipulation, cursor control and transaction management statements.
- PL/SQL improves server performance by reducing the number of calls from your application to Oracle.
- Your application can pass numerous SQL statements to Oracle at once, which improves network traffic and throughput.
- All PL/SQL code is portable to any operating system and platform on which Oracle runs.
- Recompilations are minimized because packages do not need to be recompiled if you redefine procedures within the package.
- Disk I/O is reduced because related functions and procedures are stored together.
- PL/SQL enables conditional, iterative, and sequential control structures, thus providing tremendous programming flexibility.
- Modularity is promoted because PL/SQL lets you break an application down into manageable, well-defined logic modules.
- Error detection and handling is easy.

## 2 ▲ Essentials of Database Management Systems

Without PL/SQL, the Oracle server must process each SQL statement individually. Each statement results in high performance overhead due to the additional call for each segment. With PL/SQL, an entire block of statements can be sent to Oracle at one time. This reduces the network traffic. PL/SQL can also add procedural processing power to Oracle tools such as Developer/2000, to improve performance. PL/SQL is composed of blocks, control structures, attributes, variables and cursors.

### C.1.1 PL/SQL Blocks

PL/SQL is a block-structured language with procedural and error-handling capabilities. These blocks are composed of procedures, functions and unnamed blocks that are logically grouped together to solve a specific problem. A PL/SQL block has three parts in it as shown in Figure 1:

```
DECLARE
.....
Declarations
.....
BEGIN
.....
Statements
.....
EXCEPTION
.....
Error Handling Statements
.....
END;
```

Figure 1 PL/SQL Block

The parts and their functions are given below:

**Declaration** – All objects of the block are declared.

**Execution** – The objects are defined to manipulate data.

**Exception** – Error-handling logic is placed here.

### C.2 PL/SQL ARCHITECTURE

PL/SQL is a technology and an integrated part of Oracle. It has its own processing engine that executes PL/SQL blocks and subprograms. This engine can be installed in the server or in any of the development tools such as Oracle Forms.

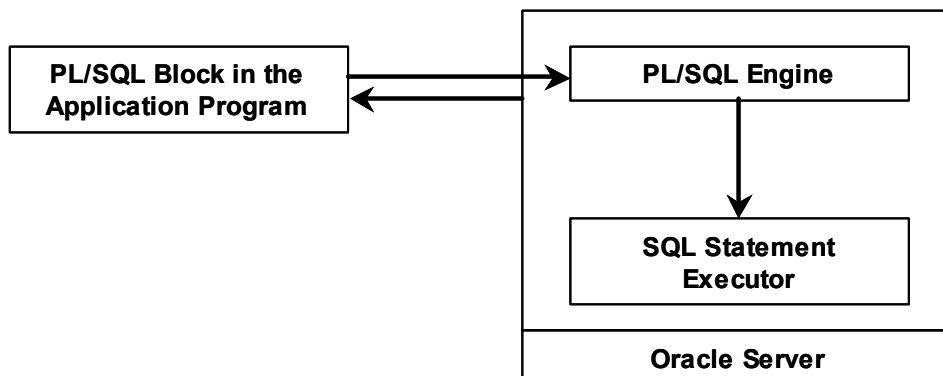


Figure 2 PL/SQL Engine

If the host program does not have a PL/SQL engine, the blocks of code are sent to the Oracle server for processing. Figure 2 shows the PL/SQL engine as an integrated component of Oracle server. The engine executes procedure statements and passes SQL statements to the SQL statement executor in the Oracle server.

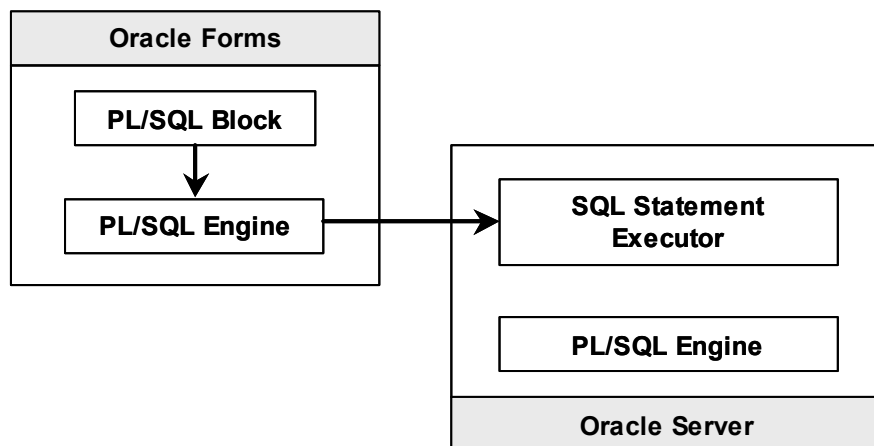


Figure 3 Processing of PL/SQL blocks by PL/SQL Engine

When Oracle tools come with a PL/SQL engine, the tools process the PL/SQL blocks directly. The Figure 3 illustrates how Oracle Forms processes PL/SQL blocks directly with its own PL/SQL engine and passing the SQL statements to the Oracle server for processing.

### C.2.1 SQL Support

By definition, PL/SQL is SQL's procedural language extension. PL/SQL supports all of SQL's data manipulation language commands (except EXPLAIN PLAN), transaction control commands, functions, pseudo-columns, and operators. PL/SQL does not support the SQL's DDL commands, session control commands or system control commands.

PL/SQL lets you take advantage of Oracle power and ease of use. In PL/SQL you can access remote databases, manage transactions, customize the optimizer, manage cursors, use distributed global names and use triggers.

### C.3 PL/SQL VARIABLES

PL/SQL is similar to any other programming language in that it has its own structure, syntax and semantics. You can define variables and use them in SQL and procedural statements anywhere an expression can be used. The only restriction is that forward references are not allowed; that is you must declare all variables and constants before referencing them in other statements.

PL/SQL variables have the same data types as SQL including CHAR, LONG, DATE, NUMBER and VARCHAR2. Additionally, PL/SQL has several of its own data types including BOOLEAN and BINARY\_INTEGER. The following example shows how the variables and constants are defined:

```

part_nbr VARCHAR2(20);
status BOOLEAN;
  
```

You can assign values to variables using the assignment operator (: =) as follows:

```

price := qty* (cost*5/2);
status := TRUE;
  
```

## 4 ▲ Essentials of Database Management Systems

A second way to assign a value to a variable is to use the SELECT statement to fetch a value into the variable as shown below:

```
SELECT name INTO var_emp
FROM employee
WHERE emp_no := var_empno;
```

To declare a constant include the keyword CONSTANT in the declaration as follows:

```
cost := CONSTANT real := 2.3;
```

### C.3.1.1 Variable and Constant Attributes

Each PL/SQL variable and constant has attributes associated with it. These attributes are properties of the variable or constant that you can reference. Following are the PL/SQL attributes:

#### C.3.1.2 %TYPE

%Type describes the data type of the variable, constant or table column. You can declare a variable using the %type attribute as shown below:

```
part_nbr := inventory.part_id%type;
```

In the above example, the variable part\_nbr is declared with the same data type as is defined for the column part\_id in the INVENTORY table.

#### C.3.1.3 %ROWTYPE

%Rowtype describes a record type that represents a row in a table. The following example creates a record number named part\_rec that has fields with the same name and data types as the columns that appear in the INVENTORY table:

```
DECLARE
part_rec inventory%rowtype;
```

## C.3.2 The Character Set

The PL/SQL character set includes the following:

- The alphabets, upper and lower case. PL/SQL is not case sensitive except within character and string literals.
- Numbers 0 to 9
- Spaces, tabs and carriage returns
- Special symbols ( ) + - \_ \* / & ^ % \$ # @ ! : ; { } [ ] ? \ > < |

## C.3.3 PL/SQL Sentence Structure

Every sentence of PL/SQL text is structured to improve readability and enforce PL/SQL's syntax.

```
SELECT part_name FROM inventory WHERE part_id > (cost *2.5); -- high value items
```

The above PL/SQL sentence includes the following:

- Symbols - \* and >. They have pre-defined meanings.
- Labels or identifiers - part\_name, inventory, part\_id and cost are names of database objects.
- Literals - 2.5
- Comments - Anything after -- is a comment.

### C.3.4 Comments

Comments are essential for any program. They improve the readability and maintainability of the program. Oracle ignores the comments. The single line comment begins with the double hyphen (--). This comment indicator can begin anywhere on a line and continues to the end of the line. The multi-line comment indicator begins with /\* and end with \*/. This comment can span multiple lines. For example:

```
/* The following section selects all the high value items from the inventory
table. High value items are those items whose cost is greater than Rs. 2000.*/
SELECT part_name FROM inventory WHERE cost > 2000;
```

The use of comments has the following restrictions:

- You cannot nest comments
- You cannot use single-line comments if the block is processed dynamically by a precompiler program.

### C.4 PL/SQL DATA TYPES

PL/SQL variables and constants have data types. The data type specifies a storage format, constraint and a valid range of values. The following are the PL/SQL data types:

- **Scalar data types (Numeric Data types)** – BINARY\_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER, NATURAL, NATURALN1, NUMBER, NUMERIC, PLS\_INTEGER, POSITIVE, POSITIVEN, REAL, SIGNTYPE and SMALLINT
- **Scalar data types (Character Data types)** – CHAR, CHARACTER, LONG, NCHAR, NVARCHAR2, STRING, VARCHAR and VARCHAR2
- **Scalar data types (Boolean Data types)** – BOOLEAN.
- **Scalar data types (Date Data types)** – DATE
- **Scalar data types (Raw Data types)** – RAW, LONG RAW
- **Scalar data types (Rowid Data types)** – ROWID
- **Composite Data types** – RECORD, TABLE, VARRAY
- **Reference Data types** – REF CURSOR, REF object type
- **LOB Data types** – BFILE, LOB, CLOB, NLOB

#### C.4.1 Scalar Data Types

The scalar data types can be divided into families like numeric, character, raw, date, rowid, etc. The numeric family stores integer or real values. There are three basic types—NUMBER, PLS\_INTEGER and BINARY\_INTEGER. Variables of type NUMBER can store both integer and real values whereas BINARY\_INTEGER and PLS\_INTEGER can hold only integers.

The variables of the character family are used to hold strings or character data. The data types in the character family are VARCHAR2, CHAR, LONG, NCHAR and NVARCHAR2. The data types of the raw family are used to store binary data. The date family is used to store data and time information. The rowid family is used to store the rowid.

##### C.4.1.1 NUMBER

NUMBER data type can hold a numeric value, either integer or floating point. The syntax for declaring number is NUMBER (P,S) where P is the precision and S is the scale. Precision is the number of digits in the value and scale is the number of digits right of the decimal point.

So a data type NUMBER (7,2) means that the data type can have seven digits out of which 2 will be decimal points. Both precision and scale are optional, but if scale is present, then precision must also be present. DEC, DECIMAL, DOUBLE PRECISION, INTEGER, INT, NUMERIC, REAL and SMALLINT are all alternate names of NUMBER (P, S). Some examples of number data type are:

## 6 ▲ Essentials of Database Management Systems

- 1234.5678
- 1234.44
- 123
- 123.4567
- 1.2345

### C.4.1.2 BINARY\_INTEGER

The NUMBER data type is stored in a decimal format. This format is optimized for accuracy and storage efficiency. Because of this, arithmetic operations cannot be performed directly on the NUMBER data type. In order to perform numeric calculations, the NUMBER data type is converted into BINARY data type.

Table 1 BINARY\_INTEGER Subtypes

Subtype	Values the subtype can hold
NATURAL	0 – 2147483647
NATURALN	0 – 2147483647 NOT NULL
POSITIVE	1 – 2147483647
POSITIVEN	1 – 2147483647 NOT NULL
SIGNTYPE	-1, 0, 1

This conversion is done automatically by the PL/SQL engine. But if you have values, which will not be stored in a database and are used only for computational purposes the BINARY\_INTEGER data type can be used to improve performance. The BINARY\_INTEGER data type can hold signed integer values in the range –2147483647 to +2147483647. The numbers are stored in a 2's compliment format, which means that they are available for computations without conversion. There are five subtypes for BINARY\_INTEGER, but they have limitations on the values they can hold. These subtypes and the values that they can hold are given in the following Table 1.

### C.4.1.3 PLS\_INTEGER

PLS\_INTEGER have the same range as BINARY\_INTEGER (–2147483647 to +2147483647). They are also implemented using the 2's compliment format. But if a calculation, which involves a PLS\_INTEGER, overflows, an error is raised. But if a calculation that involves a BINARY\_INTEGER overflows, the result can be assigned to a NUMBER variable, which has a greater range with no error.

### C.4.1.4 VARCHAR2

This data type is the same as the VARCHAR2 database type. VARCHAR2 type variables can hold variable length character strings with a maximum length of 32,767 bytes. The syntax for declaring the VARCHAR2 variable is VARCHAR2(L), where L is the maximum length of the variable. The length value must be specified, as there is no default. The length of the VARCHAR2 is specified in bytes, not in characters. The subtype of VARCHAR2 is VARCHAR, which is the same as VARCHAR2. Note that the VARCHAR2 database column can hold only 4000 bytes (in Oracle 8) and if a VARCHAR2 PL/SQL variable is more than 4000 bytes, it can be only inserted into a database column of type LONG, which has a maximum length of 2GB.

### C.4.1.5 CHAR

Variables of this type are fixed-length character strings. The syntax for declaring a CHAR variable is CHAR(L), where L is the length in bytes. If the length is optional and if it is not specified then it will default to 1. So CHAR is the same as CHAR(1). The maximum length of CHAR variable is 32767 bytes. The maximum length of the CHAR database column is 2000 bytes (in Oracle 8). Therefore if a CHAR of a PL/SQL variable is more than 2000 bytes, it can be only inserted into a database column of type VARCHAR2 or LONG.

### C.4.1.6 LONG

The PL/SQL LONG data type is a variable length string with a maximum length of 32760 bytes. LONG variables are very similar to VARCHAR2 variables.

### C.4.1.7 NCHAR and NVARCHAR2

NCHAR and NVARCHAR2 are provided in Oracle 8 and is available in PL/SQL version 8.0. These are National Language Support (NLS) character types CHAR and VARCHAR2. NLS are used to store character strings in different character sets from the PL/SQL language itself.

### C.4.1.8 RAW

RAW variables are similar to CHAR variables, except that they are not converted between character sets. The syntax for specifying RAW variable is RAW(L), where L is the length in bytes of the variable. This data type is used to store fixed length binary data. The maximum length of a RAW variable is 32767 bytes. The maximum length of a RAW database column is 255 bytes and hence if the length of the RAW variable is more than 255 bytes it cannot be inserted into a RAW database column.

### C.4.1.9 LONG RAW

RAW variables are similar to LONG variables, except that they are not converted between character sets. The maximum length of a LONG RAW variable is 32760 bytes. The maximum length of a LONGRAW database column is 2GB and hence there are no problems in inserting it into a LONG RAW database column.

### C.4.1.10 DATE

The DATE data type in PL/SQL behaves the same way as the DATE database type. The DATE type is used to store date and time information. This includes century, year, month, day, hour, minute and second. A DATE variable is 7 bytes, one byte for each component (century through second). Values are assigned to the date variable using the TO\_DATE function. The TO\_CHAR function can convert a date character.

### C.4.1.11 ROWID

The ROWID type is the same as the database ROWID pseudocolumn type. It can hold a ROWID, which can be thought of as a unique key for every row in the database. ROWIDs are stored internally as a fixed-length binary quantity.

### C.4.1.12 BOOLEAN

BOOLEAN variables are used in PL/SQL control structures IF-THEN-ELSE and LOOP statements. A BOOLEAN value can hold TRUE, FALSE or NULL only.

## C.4.2 Composite Types

The three composite types available in PL/SQL are RECORD, TABLE and VARRAY. A composite type is one that has components within it. A variable of a composite type contains one or more scalar variables.

### C.4.2.1 RECORDS

A record represents a row in a table or a row fetched by a cursor. Records have uniquely named fields and can belong to different data types. To declare a PL/SQL record, you first declare a table record type and then declare variables of that type.

```
DECLARE
TYPE emp_rec_type IS RECORD
(emp_no      NUMBER(5)      NOT NULL,
name        VARCHAR2(50),
dob         DATE);
```

Once declared, you can declare records of that type:

```
emp_rec      emp_rec_type
```

The following example shows the declaration and usage of PL/SQL records:

```
DECLARE
TYPE emp_rec_type IS RECORD
```

## 8 ▲ Essentials of Database Management Systems

```
(emp_no      NUMBER(5)      NOT NULL,  
name        VARCHAR2(50),  
dob         DATE);  
emp_rec     emp_rec_type;  
BEGIN  
SELECT empno, name, d_dob  
INTO emp_rec  
FROM employee;  
END;
```

The following rules apply when using PL/SQL records:

- Records of different types cannot be assigned to each other.
- You cannot assign a list of values to a record by using the assignment statement.
- Records cannot be tested for equality or inequality.

### C.4.2.2 TABLES

PL/SQL tables are objects of type TABLE, which look similar to database tables but aren't quite the same. These tables give you an array-like access to rows of data. PL/SQL tables are declared in the declaration portion of the block. They contain one column and a primary key, neither of which can be named. The primary key must be defined with the BINARY\_INTEGER data type. To declare PL/SQL table, you first declare a table type and then declare variables of that type.

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

Once the table is declared, declare variables of that type. For example, the following declares variables using the table type defined earlier:

```
emp_table    emp_table_type
```

To reference a PL/SQL table, specify a primary key value using the array like syntax. For example, to reference the 18th row in the PL/SQL table created in the previous example:

```
emp_table(18);
```

Taking this example a little further, you could assign values directly to a row in the PL/SQL table.

```
DECLARE  
TYPE emp_table_type IS TABLE OF VARCHAR2(30)  
INDEXED BY BINARY_INTEGER;  
k      BINARY_INTEGER := 0;  
BEGIN  
FOR new_emp IN (SELECT emp_name FROM employee  
                WHERE dept = 'ADMN');  
  
LOOP  
k := k + 1;  
emp_table(k) := new_emp.emp_name -- insert row into PL/SQL table  
END LOOP;  
END;
```

The following rules apply when using PL/SQL tables:

- A loop must be used to insert values from a PL/SQL table into a database column.
- A loop must be used to fetch data from a database column in a PL/SQL code.
- You cannot use the DELETE command to delete the contents of a PL/SQL table. You must assign an empty table to the PL/SQL table being deleted.

### C.4.2.3 VARRAY

A VARRAY is a variable length array and is a data type, which is very similar to an array in C or Pascal. Syntactically, a VARRAY is accessed in the same way as a nested or index-by table. However, a VARRAY is implemented differently. Rather than being a sparse data structure with no upper bound, elements are inserted into a VARRAY starting at index 1, up to the maximum length declared in the VARRAY type. Storage for a VARRAY is the same as a C or Pascal array, as opposed to the storage for a nested table, which is more like a database table. A VARRAY is declared using the syntax

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (maximum_size) OF element_type [NOT NULL];
```

'type\_name' is the name of the new VARRAY type, 'maximum\_size' is the integer specifying the maximum number of the elements in the VARRAY, and 'element\_type' is a PL/SQL scalar, record or object type. The 'element\_type' can be specified using %type as well, but cannot be BOOLEAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR, TABLE or another VARRAY type.

### C.4.3 Reference Type

Once a variable of a scalar or composite type is declared in PL/SQL, the memory storage for this variable is allocated. The variable names this storage and is used to refer to it later in the program. However, there is no way to deallocate the storage and still have the variable available—the memory is not freed until the variable is no longer in scope. A reference type does not have this restriction. A reference type in PL/SQL is same as a pointer in C. A variable that is declared of a reference type can point to different storage locations over the life of the program. The reference type variables available in PL/SQL are REF CURSOR and REF object type.

### C.4.4 LOB Types

The LOB types are used to store large objects. A large object can be either a binary or character value upto 4 GB in size. Large objects can contain unstructured data, which is accessed more efficiently than LONG and LONG RAW data, with fewer instructions.

## C.5 PL/SQL PRECOMPILERS

Oracle permits you to embed PL/SQL blocks within programs written in any of the following languages:

- C
- COBOL
- FORTRAN
- Ada
- Pascal
- PL/I

After writing your program in any of these languages, you precompile the source code. This precompilation checks for errors and creates a modified source code file that can be compiled, linked and executed. To embed a PL/SQL block in your program, you must precede the block with EXEC SQL EXECUTE statement. The PL/SQL block should end with the END\_EXEC command.

## 10 ▲ Essentials of Database Management Systems

```
EXEC SQL EXECUTE  
PL/SQL block  
END-EXEC
```

Use host variables to pass values to and from the host program and the PL/SQL block. Variables declared inside the PL/SQL block are global in scope. All references to host variables in a PL/SQL block must be prefaced with a colon. This is to enable the precompiler to distinguish between host and PL/SQL variables. Oracle enables you to use an optional indicator variable when referencing a host variable. This indicator variable is an integer variable that indicates the value or condition of a host variable. The indicator variable is used to detect nulls and truncate values in output host variables and to assign nulls to input variables. For output variables, the following values can be assigned to the indicator variable:

- -1 - This indicates to the host program that the value of the variable is null.
- 0 - Oracle passes the value of the variable to the host program without modification.
- >0 - Oracle assigns a truncated value to the host variable.

For input variables, the following values can be assigned to host variables by your program:

- -1 - Oracle ignores the value of the input host variable and assigns a null to it.
- >=0 - Oracle accepts the value of the host variable without modification.

The indicator variable must be declared as a 2-byte integer. In use with an SQL statement, the variable must be prefaced with a colon and appended to its host variable.

```
-- PL/SQL block  
EXEC SQL EXECUTE  
BEGIN  
SELECT qty, price, name  
INTO :ws_qty, :ws_price, :ws_name  
FROM inventory  
END;  
END-EXEC  
  
-- Host program logic  
IF ws_qty = -1  
THEN  
statements  
ELSE  
statements
```

## C.6 CONDITIONAL AND SEQUENTIAL CONTROL STATEMENTS

With PL/SQL, you can use SQL statements to manipulate Oracle data and flow control statements to process data to meet all your requirements. PL/SQL combines the data-manipulation power of SQL with the data processing power of procedural languages. PL/SQL provides many different structures to control the flow of your statement execution and data handling. These structures include the following:

- Cursors
- Procedures
- Functions
- Packages

Control structures are the most important PL/SQL extensions to SQL. Not only does PL/SQL let you manipulate data but also it lets you process the data using conditional, iterative, sequential and unconditional flow-control statements such as IF-THEN-ELSE, FOR-LOOP, WHILE-LOOP, EXIT-WHEN and GOTO. Collectively these statements can handle any situation. Additionally, PL/SQL provides benefits in the following areas:

- Improved productivity by adding functionality to non-procedural tools such as Oracle Reports and Forms.
- Better performance by enabling a block of SQL statements to be transmitted through a network to the server all at one time.
- Portability
- Integration of both PL/SQL and SQL enabling you to benefit from both

### C.6.1 Simple Blocks

PL/SQL is block-structured language. Blocks are most fundamental logical units of work that make up a PL/SQL program. These blocks include such structures as procedures, functions and cursors. Blocks can themselves contain other blocks or nested blocks. Blocks of PL/SQL code are created to solve a specific task or problem. You can have a PL/SQL block that simply counts the number of employees who have attended office every day. Blocks are made up of three distinct parts:

- Declaration and initialization of variables
- Executable PL/SQL code
- Exception Handler

The declaration portion of the block accomplishes the following:

- Declares the name of an identifier
- Declares whether the identifier is a variable or constant
- Establishes the data type for the identifier
- Establishes whether the identifier is nullable
- Initializes the identifier, if applicable

The following is a typical PL/SQL block:

```

DECLARE
    new_emp_no          empno%type;
    new_emp_name       empname%type;

BEGIN

    SELECT empno
    INTO new_emp_no
    FROM employee_table
    WHERE empname = :ws-emp_name;
    COMMIT;

    EXCEPTION
    WHEN NO_DATA_FOUND THEN
        ROLLBACK;
    COMMIT;

END;
```

## C.7 CONTROL STRUCTURES

Control structures enable the program to change the logical flow of statements within PL/SQL with a number of control structures. PL/SQL supports four basic programming control structures:

- Conditional
- Iteration
- Sequential
- Unconditional Branching

These four structures can be combined in any way to solve any given problem.

### C.7.1 Conditional Control

You can execute a statement sequence or sequence of statements conditionally with the IF-THEN-ELSE statement. Sometimes it is necessary to take alternative actions, depending on a condition or circumstance.

#### C.7.1.1 IF-THEN Statements

The simple form of the conditional control statement is the IF-THEN statement. The syntax for the IF-THEN statement is as follows:

```
IF condition THEN
    statement(s)
END IF;
```

The statements that immediately follow the conditional IF statement are executed only if the condition evaluates to True. If the condition evaluates to False, the statements are not processed and are passed over.

#### C.7.1.2 IF-THEN-ELSE Statements

The syntax of the IF-THEN-ELSE statement is:

```
IF condition THEN
    statement(s)
ELSE
    statement(s)
END IF;
```

The statements that immediately follow the conditional IF statement are executed only if the condition evaluates to True. If the condition evaluates to False, the statements immediately following the ELSE portion are executed. You can have nested IF-THEN-ELSE statements if you want.

```
IF grade = 'A' THEN
    UPDATE employee_table
    SET tax = salary*0.4
ELSE
    IF grade = 'B' AND salary >=5000
        UPDATE employee_table
        SET tax = salary*0.3
    ELSE
        UPDATE employee_table
```

```

        SET tax = salary*0.25
    END IF
END IF;

```

Each IF-THEN statement can have only one ELSE statement and must end with an END IF statement.

### C.7.1.3 IF-THEN-ELSEIF Statements

In some situations you need to select an action from several mutually exclusive alternatives. The IF-THEN-ELSEIF statement gives a solution for this. The IF-THEN-ELSEIF statement enables you to have several mutually exclusive conditions. The syntax for this statement is as follows:

```

IF condition1 THEN
    statement(s);
ELSEIF condition2 THEN
    statement(s);
ELSE
    statement(s)
END IF;

```

This is how this structure works. PL/SQL evaluates the first condition. If it is True, the statements following that are executed and the control passes to the statement after the END IF. If condition1 evaluates to False, the ELSEIF clause is tested. If it is True the statements after it are executed and control goes to statement after the END IF. If the condition2 is found False, then control goes to the statements after the ELSE clause.

You have only one ELSE clause in this structure. But you can have as many ELSEIFs as you want. For clarity you can use ELSEIF clause instead of nested IFs.

## C.7.2 Sequential control

By default, all PL/SQL blocks are executed in a top-down sequential process. The process begins with the BEGIN statement and terminates with the END statement. The developer can deviate from this default control structure by introducing one of the control structures discussed.

## C.7.3 Unconditional Control

Two of the least-used control structures are the GOTO and NULL unconditional statements. The GOTO statement forces the flow of processing to branch to a label unconditionally. When executed, the GOTO statement transfers control to the labeled statement or block. The syntax of the GOTO statement is:

```

GOTO label_name;

```

In the following example, the GOTO statement is used to transfer processing control to the label new\_employee:

```

SELECT COUNT(*), empno INTO emp_count
FROM employee
FOR counter 1..emp_count LOOP
SELECT empno FROM salary;
IF empno > 10000THEN
    GOTO new_employee;
ELSE
    INSERT INTO emp_audit VALUES (:user_name, empno);
END IF;

```

## 14 ▲ Essentials of Database Management Systems

```
<< new_employee>>
INSERT INTO employee VALUES (empno);
INSERT INTO emp_audit VALUES (:user_name, empno);
END LOOP;
```

GOTO Statements cannot do the following:

- Branch into an IF or LOOP statement or sub-block.
- Branch out of a subprogram
- Branch out of an exception handler in the current block.

The NULL statement does nothing other than pass control to the next statement. Specifically the NULL statement means inaction.

```
IF price <= 50 THEN
    statements
ELSE
    NULL -- do nothing
END IF;
```

### C.8 CURSORS

Cursors are constructs that enable the user to name a private memory area to hold a specific statement for access at a later time. There are two kinds of cursors used in Oracle: implicit and explicit. PL/SQL implicitly declares a cursor for every SQL data manipulation statement, such as INSERT, UPDATE and DELETE. Explicit cursors are declared by the user and are used to process query results that return multiple rows.

The number of rows returned by a query can be zero, one or more than one. Oracle provides the cursor as a mechanism to be used on such conditions. Oracle provides the cursor as the mechanism to be used to easily process these multiple row result sets one row at a time. Without cursors, the Oracle developer will have to explicitly fetch and manage each individual row that is selected by the query.

The multiple rows returned from a query are called the active set. PL/SQL defines its size as the number of rows that have met your search criteria and formed the active set. Inherent in every cursor is a pointer that keeps track of the current row that is being accessed, which enables your program to process the rows one at a time.

You can view a cursor as a file that needs to be processed from the top to the bottom of the file, one record at a time. Cursors are used to process multi-row result sets one row at a time. Additionally, cursors keep track of which row is currently being accessed, which allows for iterative processing of the active set.

Using several cursors simultaneously can improve system performance by reducing the frequency of parsing and cursor opening. After each iteration of the cursor, the cursor retains enough information about the active set and the SQL statements that it can be re-executed many times. Each iteration is accomplished without having to re-open and parse the cursor every time.

By opening several cursors, the parsed representation of the SQL statements can be saved, thereby eliminating the repeated cost of opening the cursor and parsing.

#### C.8.1 Processing Explicit Cursors

Following steps are required to create and use explicit cursors:

- Declare the cursor
- Open the cursor

- Fetch the data
- Close the cursor

### C.8.1.1 Declaring a cursor

Declaring a cursor accomplishes two goals: naming the cursor and associating the query with the cursor. The cursor can be declared using the DECLARE CURSOR command. The syntax for declaring the cursor is:

```
CURSOR cursor_name IS SELECT statement;
```

Here a point to be noted is that since the cursor name is a PL/SQL identifier, it must be declared before it is referenced. In the following statement a cursor named c\_emp is declared.:

```
CURSOR c_emp IS
SELECT * FROM employee_table;
```

### C.8.1.2 OPEN, FETCH and CLOSE

Opening the cursor executes the query and identifies the active set. The active set contains all the rows that meet the query search criteria. The cursor can be opened using the OPEN command followed by the cursor name. The FETCH command is used to retrieve the rows in the active set one at a time. The first FETCH statement sorts the active set, if necessary. The cursor advances to the next row in the active set each time the FETCH command is executed. You can move through the active set only with the FETCH command. The contents of the FETCHed rows can be assigned to host variables or to a RECORD data type. After doing the data manipulations the cursors should be closed. This can be achieved by using the command CLOSE followed by the cursor name.

### C.8.1.3 Cursor Attributes

There are four attributes available in PL/SQL that can be applied to cursors. Cursor attributes are appended to a cursor name in a PL/SQL block, similar to the %TYPE and %ROWTYPE. But instead of returning a type, the cursor attributes return values that can be used in expressions. The cursor attributes are %FOUND, %NOTFOUND, %ISOPEN and %ROWCOUNT.

- **%FOUND** – %FOUND is a Boolean attribute. It returns TRUE if the previous FETCH returned a row and FALSE if it didn't. If %FOUND is checked while the cursor is closed the ORA-1001 (invalid cursor) is returned.
- **%NOTFOUND** – %NOTFOUND is a Boolean attribute. It behaves opposite to the %FOUND. It returns FALSE if the previous FETCH returned a row. %NOTFOUND returns TRUE only if the prior FETCH does not return a row. As shown in the following example, it is often used as an exit condition for a fetch loop. If %FOUND is checked while the cursor is closed the ORA-1001 (invalid cursor) is returned.
- **%ISOPEN** – This also is a Boolean attribute and is used to determine whether a cursor is open or not. If the cursor is open %ISOPEN returns TRUE otherwise it will return FALSE.
- **%ROWCOUNT** – This numeric attribute returns the number of rows fetched by the cursor so far. If %ROWCOUNT is checked while the cursor is not open the ORA-1001 (invalid cursor) is returned.

An example of a cursor is given below. It declares a cursor c\_emp for the table employee and fetches the salary and deductions of all the employees, calculates the net salary for each employee and then inserts those values into the salary table:

```
DECLARE
CURSOR c_emp IS
SELECT empname, salary, deductions FROM employee_table;
c_name          VARCHAR2(20);
c_salary       NUMBER(7,2) := 0;
```

## 16 ▲ Essentials of Database Management Systems

```
c_deductions      NUMBER (7,2) : =0;
net_salary        NUMBER(7,2) : =0;

BEGIN

    OPEN c_emp
    LOOP
        FETCH c_emp INTO c_name,c_salary, c_deductions;
        EXIT WHEN c_emp%NOTFOUND;
        net_salary = c_salary - c_deductions;
        INSERT INTO salary_table VALUES (c_name, net_salary);
    END LOOP;
    CLOSE c_emp;
    COMMIT;

END;
```

### C.9 ITERATIVE CONTROL STATEMENTS

Iterative control statements enable you to execute a sequence of statements multiple times. The simplest form of an iterative statement is the loop.

#### C.9.1 Loop Statements

The syntax of the loop statement is

```
LOOP
    statements(s)
END LOOP;
```

Each time the loop is processed, control is always resumed at the top of the loop structure and statements are executed. The EXIT statement provides a way to stop the iterative loop. The EXIT statement must be placed inside the loop.

```
BEGIN

    LOOP

        count = count +1
        IF count >=14 THEN
            EXIT;
        END IF;
    END LOOP;

END;
```

#### C.9.2 FOR LOOP Statements

The FOR LOOP is a mechanism to cycle through a loop fixed number of times. A FOR LOOP iterates over and over for a specified number of times. The syntax for the FOR LOOP is:

```
FOR counter IN lower bound . . upper bound LOOP
    statement(s)
END LOOP;
```

The range of valid integers for the counter is evaluated when the loop is first entered and is never re-evaluated. After each iteration, the counter is implicitly incremented. Also the counter is implicitly

declared, so you do not have to declare it. If the upper bound value is less than the lower bound value the loop is never executed. In the following loop the statements will be executed 30 times:

```
FOR k IN 1..30 LOOP
    Statements
END LOOP;
```

The lower and upper bound values can be literals, variables or expressions, but they must evaluate to integers. Additionally the loop range can be determined dynamically at runtime as shown in the following example, where the upper bound of the loop is equal to the number of employees in the company:

```
SELECT COUNT(*) INTO u_counter
FROM employee;
FOR k IN 1..u_counter LOOP
    UPDATE employee
    SET salary = salary * 1.1;
END LOOP;
```

You can also include the REVERSE clause to decrement the counter.

```
FOR k IN REVERSE 1..10 LOOP
    statements
END LOOP;
```

In the above example the counter begins at 10 and is decremented by 1 each time the loop is executed. When the loop is finished, the counter variables become undefined. Also, while the loop is active the counter variables are local and cannot be referenced outside the loop.

### C.9.2.1 LOOP Labels

In PL/SQL, you may attach a label name to a loop. This label, which is an undeclared identifier, must appear at the beginning of the loop and must be enclosed in double angle brackets. Optionally this label may be placed at the end of the looping statement without angle brackets. The following example uses labels to clearly mark the beginning and end of the looping structure:

```
<< salary_loop>> -- loop label
FOR k IN 1..u_counter LOOP
    UPDATE employee
    SET salary = salary * 1.1;
END LOOP inventory_loop;
```

### C.9.3 WHILE Statements

The WHILE -LOOP structure repeats a statement until a condition is no longer true. The syntax is:

```
WHILE condition LOOP
    statement(s)
END LOOP;
```

Before each iteration, the condition is evaluated. If the condition is True, the statements are executed. Otherwise the statements are skipped and not executed.

```
DECLARE
counter    NUMBER(2) := 1;
```

## 18 ▲ Essentials of Database Management Systems

```
BEGIN
    WHILE counter >20 LOOP
        SELECT name INTO guest_name
        FROM register
        counter = counter +1;
    END LOOP;
END;
```

The number of iterations that will be performed by a WHILE loop is sometimes unknown till the loop ends. It depends on the condition statement. Because the condition is evaluated at the top or beginning of the loop, the statements might not even be executed.

### C.10 PL/SQL EXCEPTIONS

The Oracle server and the PL/SQL engine generate error messages and codes for every runtime execution error or hardware failure. The format of the error message is:

#### PLS-00000 error message

The error code is prefixed by PLS and followed by a unique number. By using this number, more details about the error can be found from the 'PL/SQL User's Guide and Reference' or 'The Oracle Messages and Codes Manual'.

Like most programs, a PL/SQL program normally stops processing when an error is detected. An exception to this is when you take advantage of PL/SQL's exception-handling capabilities. With this, your program does not have to stop processing when an error occurs. You can program your code to handle errors so that processing continues even after runtime errors are detected. PL/SQL enables the users to define exceptions on their own.

#### C.10.1 User-defined Exceptions

A user-defined exception is an error that is defined by the program. The error that it signifies need not be an Oracle error. It could be an error with data such as an invalid data being entered or something like that. Predefined errors correspond to SQL errors. User-defined exceptions are declared in the declarative section of a PL/SQL block. Just like variables, errors have a type—EXCEPTION. An example of an exception declaration is given below:

```
DECLARE
    e_invalid-date    EXCEPTION;
```

The user-defined exceptions have to be declared and must be raised explicitly by a RAISE command. We have seen how to declare the exception. Now we will see an example of the RAISE command.

```
DECLARE
    e_invalid_data EXCEPTION;
    v_empno NUMBER(3);
    v_name CHAR(20);
    v_sex CHAR(1);

BEGIN
    SELECT empno, name, sex
    INTO v_empno, v_name, v_sex
    FROM employee
```

```

WHERE empno = 111;
IF v_sex = 'M' THEN
    statement(s);
ELSEIF v_sex = 'F' THEN
    statement(s);
ELSE
    RAISE e_invalid_data;
END IF;
END

```

When an exception is raised, control immediately passes to the exception section of the block—section where the code is written for handling the exceptions. The exception section consists of handlers for all the exceptions. An exception handler contains the code that is executed when the error associated with it occurs and the exception is raised. The syntax of the exception section is given below:

```

EXCEPTION
WHEN exception_name THEN
    error handling statements
WHEN exception_name THEN
    error handling statements
.....
.....
WHEN OTHERS THEN
    error handling statements
END;

```

The exception handler will execute the appropriate error handling statements depending on the name of the exception that is being raised. If the exception that is being raised is not mentioned in the EXCEPTION block then the OTHERS exception handler will be used and the statements associated with it will be executed. So the WHEN OTHERS... statement should be the last handler in the block.

### C.10.2 Pre-defined Exceptions

Oracle has pre-defined several exceptions that correspond to the most common Oracle errors. These exceptions are already available in the program and need not be declared explicitly. Some of the pre-defined exceptions are given in Table 2.

Table 2 Pre-defined Exceptions

Exception	Equivalent Oracle Error
INVALID_CURSOR	ORA-1001
NO_DATA_FOUND	ORA-1403
TOO_MANY_ROWS	ORA-1422
ZERO_DIVIDE	ORA-1476
CURSOR-ALREADY_OPEN	ORA-6511

In the case of the pre-defined exceptions, there is no need for explicitly raising the exception; they are raised implicitly when their associated Oracle error occurs.

## C.11 PL/SQL BLOCKS

The basic unit of any PL/SQL program is the block. All PL/SQL programs are composed of blocks, which can occur sequentially (one-after-other) or can be nested (one inside the other). The different types of PL/SQL blocks are anonymous, named, subprograms and triggers.

- Anonymous Blocks are generally constructed dynamically and executed only once.
- Named blocks are anonymous blocks with a label that gives the block a name. These are also generally constructed dynamically and executed only once.
- Subprograms are procedures, packages, and functions that are stored in the database. These blocks generally do not change once they are constructed, and they are executed many times. Subprograms are executed explicitly using a call to the procedure, package or function.
- Triggers are named blocks that are also stored in the database. They also generally do not change once they are constructed and are executed many times. Triggers are executed implicitly whenever the triggering event occurs. The triggering event can be a DML statement executed against a table in the database.

### C.11.1 Anonymous Blocks

We have seen that a PL/SQL is a block-structured language with procedural and error-handling capabilities. These blocks are composed of procedures, functions and unnamed blocks that are logically grouped together to solve a specific problem. A PL/SQL block has three parts in it—Declaration, Execution and Exception. The anonymous block also has the same structure. For example, the following anonymous block inserts the employee number, name, salary and department of a new employee into the employee table, selects the name of the employee whose employee number is 111 and displays it on the screen.

```

DECLARE
  /* Declares the variables to be used in the block; The ':=' is used to assign
  values to      the variables */
  v_empno NUMBER (3) := 125;
  v_name   CHAR (20) := 'Mathews Leon';
  v_salary NUMBER (5) := 5000;
  v_deptid CHAR (2) := 'D1';
  v_display_string CHAR (20);

BEGIN

  /* The new values are inserted into the employee table */

  INSERT
  INTO employee (empno, name, salary, deptid)
  VALUES (v_empno, v_name, v_salary, v_deptid);

  /* Note that instead of using the variables and assigning the values to it, the
  values could be specified directly into the INSERT statement. This is given in
  the following INSERT statement */

  INSERT
  INTO employee (empno, name, salary, deptid)
  VALUES (126, 'Alexis Leon', 4000, 'D2');

```

```

/* The above two statements insert two rows into the employee table. The next
statement selects a row from the employee table and displays it on the screen
using the DBMS_OUTPUT package. */

```

```

SELECT name
INTO v_display_string
FROM employee
WHERE empno = 111;

DBMS_OUTPUT.PUT_LINE(v_display_string);

EXCEPTION
WHEN NO_DATA_FOUND THEN
    INSERT INTO exception_log (details)
    VALUES ('Specified row does not exist');
WHEN OTHERS THEN
    INSERT INTO exception_log (details)
    VALUES ('Unknown error');
END;

```

In order to name this block, you just have to put a label before the DECLARE keyword. The label can be given after the END statement, but that is optional. We will make the above anonymous block into a named block with name 'sample'. This is given below:

```

<<sample>>
DECLARE
/* Declares the variables to be used in the block; The ':=' is used to assign
values to the variables */

v_empno NUMBER (3) := 125;
v_name    CHAR (20) := 'Mathews Leon';
v_salary  NUMBER (5) := 5000;
v_deptid  CHAR (2) := 'D1';
v_display_string CHAR (20);

BEGIN
/* The new values are inserted into the employee table */

INSERT
INTO employee (empno, name, salary, deptid)
VALUES (v_empno, v_name, v_salary, v_deptid);

/* Note that instead of using the variables and assigning the values to it, the
values could be specified directly into the INSERT statement. This is given in
the following INSERT statement */

INSERT

```

## 22 ▲ Essentials of Database Management Systems

```
INTO employee (empno, name, salary, deptid)
VALUES (126, 'Alexis Leon', 4000, 'D2');

/* The above two statements insert two rows into the employee table. The next
statement selects a row from the employee table and displays it on the screen
using the DBMS_OUTPUT package. */

SELECT name
INTO v_display_string
FROM employee
WHERE empno = 111;

DBMS_OUTPUT.PUT_LINE(v_display_string);

EXCEPTION
WHEN NO_DATA_FOUND THEN
    INSERT INTO exception_log (details)
    VALUES ('Specified row does not exist');
WHEN OTHERS THEN
    INSERT INTO exception_log (details)
    VALUES ('Unknown error');

END <<sample>>;
```

We can make the anonymous block into a procedure by replacing the DECLARE command by CREATE OR REPLACE PROCEDURE procedure\_name. This is shown below where a procedure named 'sample' is created. The procedure name is used after the END keyword.

```
CREATE OR REPLACE PROCEDURE sample AS
/* Declares the variables to be used in the block; The ':=' is used to assign
values to the variables */

v_empno NUMBER (3) := 125;
v_name    CHAR (20) := 'Mathews Leon';
v_salary  NUMBER (5) := 5000;
v_deptid  CHAR (2) := 'D1';
v_display_string CHAR (20);

BEGIN
/* The new values are inserted into the employee table */

INSERT
INTO employee (empno, name, salary, deptid)
VALUES (v_empno, v_name, v_salary, v_deptid);

/* Note that instead of using the variables and assigning the values to it, the
values could be specified directly into the INSERT statement. This is given in
the following INSERT statement */
```

```

INSERT
INTO employee (empno, name, salary, deptid)
VALUES (126, 'Alexis Leon', 4000, 'D2');

/* The above two statements insert two rows into the employee table. The next
statement selects a row from the employee table and displays it on the screen
using the DBMS_OUTPUT package. */

SELECT name
INTO v_display_string
FROM employee
WHERE empno = 111;

DBMS_OUTPUT.PUT_LINE(v_display_string);

EXCEPTION
WHEN NO_DATA_FOUND THEN
    INSERT INTO exception_log (details)
    VALUES ('Specified row does not exist');
WHEN OTHERS THEN
    INSERT INTO exception_log (details)
    VALUES ('Unknown error');
END sample;

```

## C.12 PL/SQL 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 or to audit changes to data. The code within a trigger, called the trigger body, is made up of PL/SQL blocks.

The execution of triggers is transparent to the user. Triggers are executed by the database when specific types of data manipulation commands are performed on specific tables. Such commands may include inserts, updates and deletes. Updates of specific columns may also be used as triggering events.

Because of their flexibility, triggers may supplement referential integrity. They should not be used to replace it. When enforcing the business rules in an application, you should first rely on the declarative referential integrity available with Oracle; use triggers to enforce rules that cannot be coded through referential integrity.

Triggers are available in any Oracle database that has installed the Procedural option. This option is part of Oracle 7.1 onwards, but not a part of the default installation for Oracle 7.0. If you are using a version of Oracle that precedes Oracle 7.1, you can install the Procedural option during database installation.

### C.12.1 Required System Privileges

In order to create a trigger on a table, you must either own the table or should have ALTER privilege for the table or should have ALTER ANY TABLE system privilege. In addition, you must have CREATE TRIGGER system privilege. To create triggers in another user's account or schema, you must have CREATE ANY TRIGGER system privilege. The CREATE TRIGGER system privilege is part of the RESOURCE role provided with Oracle.

In order to alter a trigger, you must either own the trigger or have the ALTER ANY TRIGGER system privilege. You may also alter triggers by altering the tables they are based on, which requires that you either have the ALTER privilege for that table or the ALTER ANY TABLE system privilege.

### C.12.2 Required Table Privileges

Triggers may reference tables other than the one that initiated the triggering event. For example for creating audit trails, you may insert a new record in a different audit table, every time a record is changed in the table that is being audited. In order to do this, you will need to have privileges to insert into the audit table. That is, the privileges needed for triggered transactions cannot come from roles, but must be granted directly to the creator of the trigger.

## C.13 TYPES OF TRIGGERS

There are twelve basic types of triggers. A trigger's type is defined by the type of triggering transaction and by the level at which the trigger is executed. The following sections describe these classifications.

### C.13.1 Row-Level Triggers

Row-level triggers, trigger once for each row in a transaction. For the audit trail example given above, every time a row is changed in the audited table, a new row is added in the audit table. Row-level triggers are the most common types of triggers. They are often used in data auditing applications. Row-level triggers are also useful for keeping distributed data in sync. Snapshots use row-level triggers for this purpose. Row-level triggers are created using the FOR EACH ROW clause in the CREATE TRIGGER command.

### C.13.2 Statement-Level Triggers

Statement-level triggers execute once for each transaction. For example, if a single transaction inserted 500 rows into a table then a statement-level trigger on that table will be executed only once. Statement-level triggers therefore are not often used for data related activities. They are normally used to enforce additional security measures on types of transactions that may be performed on a table. Statement-level triggers are the default type of triggers created using CREATE TRIGGER command.

### C.13.3 BEFORE and AFTER Triggers

Since triggers occur because of events, they may set to occur immediately before or after those events. Since the events that execute triggers are database transactions, triggers can be executed immediately before or after INSERTs, UPDATEs and DELETEs.

Within a trigger, you will be able to reference the old and new values involved in the transaction. The access required for the old and new data may determine which type of trigger you need. Old refers to the data as it existed prior to the transaction. Updates and deletes usually reference old values. New values are the data values that the transaction creates.

If you need to set a column value in an inserted row via your trigger, then you will need to use a BEFORE INSERT trigger in order to access the 'new' values. Using an AFTER INSERT trigger would not allow you to set the inserted value, since the row would have been already inserted into the table.

AFTER row-level triggers are frequently used in auditing applications, since they do not fire until the row has been modified. Since the row has been successfully modified, this implies that it has successfully passed the referential integrity constraints defined for that table.

### C.13.4 Valid Trigger Types

When combining the different types of triggering actions, there are twelve possible combinations:

- BEFORE INSERT row
- BEFORE INSERT statement
- AFTER INSERT row
- AFTER INSERT statement
- BEFORE UPDATE row
- BEFORE UPDATE statement
- AFTER UPDATE row
- AFTER UPDATE statement

- BEFORE DELETE row
- BEFORE DELETE statement
- AFTER DELETE row
- AFTER DELETE statement

UPDATE triggers may be dependent upon the columns being updated. In Oracle 7.0 you can have only one trigger of each type per table. Therefore, you are limited to twelve triggers per table. If a table is used as part of a snapshot and has a snapshot log associated with it, then Oracle will automatically create a series of AFTER INSERT row, AFTER UPDATE row and AFTER DELETE row triggers on the table. Thus the user will not be able to create any additional AFTER [command] row triggers on the table. This restriction is not enforced from Oracle 7.1 onwards.

### C.13.5 Trigger Syntax

The syntax for the CREATE TRIGGER command is shown below:

```
CREATE [or REPLACE] TRIGGER trigger_name
[BEFORE|AFTER]
[DELETE|INSERT|UPDATE [OF column_name]]
ON [user.] table_name
[FOR EACH ROW] [WHEN trigger_condition]
[PL/SQL block];
```

Clearly there is a great deal of flexibility in the design of a trigger. The BEFORE and AFTER keywords indicate whether the trigger should be executed before or after the triggering transaction. The DELETE, INSERT and UPDATE keywords (the UPDATE may include a column list) indicate the type of data manipulation that will constitute the triggering event.

When the FOR EACH ROW clause is used, the trigger will be a row-level-trigger, otherwise, it will be a statement-level trigger. The WHEN clause is used to further restrict when the trigger is executed. The restrictions enforced in the WHEN clause may include checks of old and new data values.

For example, suppose we want to monitor any changes to the amount that is greater than 20%. The following row-level BEFORE UPDATE trigger will be executed only if the new value of the Amount column is more than 20% its old value.

```
CREATE TRIGGER ledger_bef_upd_row
BEFORE UPDATE ON ledger
FOR EACH ROW
WHEN (NEW.Amount/OLD.Amount>1.2)
BEGIN
INSERT INTO ledger_audit
VALUES (:OLD.Action_Date, :OLD.Action, :OLD.Amount, :OLD.Item);
END;
```

The above trigger is explained below:

- **CREATE TRIGGER ledger\_bef\_upd\_row** (The trigger is created and named)
- **BEFORE UPDATE ON ledger** (This trigger applies on the 'ledger' table. It will be executed before update transactions have been committed to that database.)
- **FOR EACH ROW** (Because the for each row clause is used, the trigger will apply to each row in the transaction. If the clause is not used, then the trigger will execute only at the statement level.)

## 26 ▲ Essentials of Database Management Systems

- **WHEN (NEW.Amount/OLD.Amount>1.2)** (The when clause adds further restrictions to the triggering condition. The triggering event must not only be an update of the 'ledger' table, but also must reflect an increase of over 20% in the value of the 'Amount' column.)
- **BEGIN** – Begin the PL/SQL Block
- **INSERT INTO ledger\_audit VALUES (:OLD.Action\_Date, :OLD.Action, :OLD.Amount, :OLD.Item);** – SQL
- **END;** – End of PL/SQL Block

The PL/SQL code is the trigger body. The commands shown here are to be executed for every update of the 'ledger' table that passes the when condition. In order for this to succeed, the 'ledger\_audit' table must exist, and the owner of the trigger must have been granted privileges on that table. This particular example inserts the old values of the 'ledger' table into the 'ledger\_audit' table before the 'ledger' record is updated. The BEGIN and END statements mark the beginning and end of the PL/SQL block. When referencing the new and old keywords in the PL/SQL block they are preceded by colons (:).

### C.13.6 Combining Trigger Types

Triggers for multiple INSERT, UPDATE and DELETE commands on a table can be combined into a single trigger, provided they are all at the same level (row or statement). Consider the following example:

```
CREATE TRIGGER ledger_bef_upd_ins_row
BEFORE INSERT OR UPDATE OF Amount ON ledger
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        INSERT INTO ledger_audit
        VALUES (:NEW.Action_Date, :NEW.Action, :NEW.Amount, :NEW.Item);
    ELSE
        INSERT INTO ledger_audit
        VALUES (:OLD.Action_Date, :OLD.Action, :OLD.Amount, :OLD.Item);
    END;
```

The above example shows a trigger that is executed whenever an INSERT or an UPDATE occurs. The UPDATE portion of the trigger only occurs when the Amount column is updated and IF clause is used within the PL/SQL block to determine which of the two commands executed the trigger.

Combining trigger types in this manner may help to coordinate trigger development among multiple developers, since it helps to consolidate all of the database events on a single table.

### C.13.7 Setting Inserted Values

You may use triggers to set values during inserts and updates. For example, you may have partially denormalized your table to include derived data. Sometimes the derived columns may not be in sync with the base column. For instance you may have a column ContactName and a derived column UpperContactName which is nothing but UPPER(ContactName). Since UpperContactName is derived data, it may be out of sync with the ContactName column. That is, there may be times immediately after transactions during which UpperContactName is not equal to UPPER(ContactName). Consider an insert into the table. Unless your application supplies a value for UpperContactName during inserts, that column's value will be NULL.

To avoid this synchronization problem, you may use a database trigger. Put a BEFORE INSERT and BEFORE UPDATE trigger on the table. They will act at the row level, as shown in the following listing and will set the value for UpperContactName every time Contact name is changed.

```
CREATE TRIGGER contacts_bef_upd_ins_row
BEFORE INSERT OR UPDATE OF ContactName ON contacts
```

```

FOR EACH ROW
BEGIN
    :NEW.UpperContactName := UPPER(ContactName)
END;

```

In the above example, the trigger body determines the value for the UpperContactName by using the UPPER function on the ContactName column. This trigger will be executed every time a row is inserted into the 'contacts' table and every time the ContactName column is updated. The two columns will thus be kept in sync.

In Oracle7.0, this trigger will conflict with the auditing trigger (assuming that the contacts table is audited). This conflict arises because of the restriction on the number of triggers per table in Oracle 7.0. To remove this conflict, you must either drop one of the triggers or combine the two triggers into a single larger trigger.

### C.13.8 Maintaining Duplicated Data

The method of setting values through triggers, shown in the previous section can be combined with remote data access methods. Using triggers to duplicate data is redundant in databases that have the distributed option; snapshots serve that purpose. However, if you do not have the distributed option, you may use triggers to maintain remote copies (on the same host) of your tables. As with snapshots, you may replicate all or part of the rows in a table.

For example, you may wish to create and maintain a second copy of your application's audit log. By doing so, you will safeguard against a single application, wiping out the audit log records created by multiple applications. Duplicate audit logs are frequently used in security monitoring.

Consider a transaction\_audit table used to audit the transactions table. A second table transaction\_audit\_dup, could be created, possibly in a remote database. For this example, assume that a database link called audit\_link can be used to connect the user to the database in which the transaction\_audit\_dup resides. To automate the populating of the transaction\_audit\_dup table, the following trigger could be placed on the transaction\_audit table:

```

CREATE TRIGGER transaction_after_ins_row
BEFORE INSERT ON transaction_audit
FOR EACH ROW
BEGIN
    INSERT INTO transaction_audit_dup@audit_link
    VALUES (:NEW.Date, :NEW.Hoadb, :NEW.Hoacr, :NEW.Amount);
END;

```

This trigger executes for each row that is inserted into the transaction\_audit table. It inserts a single record into the transaction\_audit\_dup table, in the database defined by the audit\_link database link. If you are using Oracle with distributed option, then audit\_link may point to a database on a remote server. But in that case, it is better to use a snapshot instead of a trigger.

### C.13.9 Customizing Error Conditions

Within a single trigger, you may establish different error conditions. For each of the error conditions you define, you may select an error message that appears when the error occurs. The error numbers and messages that are displayed to the user are set through the RAISE\_APPLICATION\_ERROR procedure. This procedure may be called from within a trigger. The RAISE\_APPLICATION\_ERROR procedure takes two input parameters: the error number (should be between -20001 and -20999) and error message to be displayed.

The following example shows a statement-level BEFORE DELETE trigger on the contacts table. When a user attempts to delete a record from the contacts table, this trigger is executed and checks for the

## 28 ▲ Essentials of Database Management Systems

following condition: the Oracle username of the account performing the delete begins with the letters "SALES".

```
CREATE TRIGGER contacts_before_del
BEFORE DELETE ON contacts
DECLARE
not_sales_user EXCEPTION;
BEGIN
    IF SUBSTR(User,1,5)<> 'SALES' THEN
        RAISE not_sales_user;
    EXCEPTIONS
        WHEN not_sales_user THEN
            RAISE_APPLICATION_ERROR (-20002,
                'Deletion of records only allowed by Sales users');
END;
```

### C.13.10 Disabling and Enabling Triggers

Unlike declarative integrity constraints, triggers do not affect all rows in a table. They only affect transactions of the specified type and that too only when the trigger is enabled. The trigger will not affect any transactions created prior to a trigger's creation. By default, a trigger is enabled when it is created. However, there are situations in which you may wish to disable a trigger. The two most common reasons involved data loads. During large data loads, you may wish to disable triggers that would execute during the load. Disabling the triggers during data loads will dramatically improve the performance of the data load. Once the data has been loaded, you will need to manually perform the data manipulation that the trigger would have performed had it been enabled during the data load.

The second data load related reason for disabling a trigger occurs when a data load fails and has to be performed a second time. In such a case, it is likely that the data load partially succeeded and thus the trigger was executed for a portion of the data load records. During subsequent data load, the same records would be inserted. Thus it is possible that the same trigger will be fired twice for the same transaction. Depending on the nature of the transactions and the triggers, this may not be desirable. If the trigger was enabled during the failed load, then it may need to be disabled prior to the second load process. Once the data has been loaded, you will manually perform the data manipulation that the trigger would have performed.

To enable a trigger, use the ALTER TRIGGER command with the ENABLE keyword. In order to use this command you must either own the table or have ALTER ANY TRIGGER system privilege. A second method of enabling a trigger uses the ALTER TRIGGER command with the ENABLE ALL TRIGGERS clause. You cannot enable specific triggers with this command. Here also to execute this command you should either own the table or have ALTER ANY TRIGGER system privilege. You can disable a trigger using the same ALTER TRIGGER command with any of the clauses DISABLE or DISABLE ALL TRIGGERS. The privileges that you require for disabling are the same as that for enabling.

### C.13.11 Replacing Triggers

The body of a trigger cannot be altered. Only the status can be altered. To alter the body, the trigger must be re-created or replaced. When replacing a trigger, you should use CREATE OR REPLACE TRIGGER commands. Using the OR REPLACE option will maintain any grants made for the original version of the trigger. The alternative solution is dropping and re-creating the trigger, but it will drop all the grants made for the trigger.

### C.13.12 Dropping Triggers

Triggers may be dropped using the DROP TRIGGER command. In order to drop a trigger you must either own the trigger or have a DROP ANY TRIGGER system privilege.

## C.14 PROCEDURES AND PACKAGES

Sophisticated business rules and application logic can be stored as procedures within Oracle. Stored procedures—groups of SQL and PL/SQL statements—allow you to move code that enforces business rules from your application to your database. As a result the code will be stored once for all applications to use. Because Oracle supports stored procedures, the code within your application should become more consistent and easier to maintain.

You can group procedures and other PL/SQL commands into packages. There will be performance gains when using procedures. This is because of many reasons: One, the processing of complex business rules may be performed within the database and therefore by the server. In client/server applications, shifting complex processing from the application (on the client) to the database (on the server) may dramatically improve the performance. Second, since the procedural code is stored within the database and is fairly static, you may also benefit from the reuse of same queries within the database. The shared SQL Area in the System Global Area (SGA) will store the parsed versions of the executed commands. Thus, the second time the procedure is executed, it may be able to take advantage of the parsing that was previously performed, improving the performance of the procedures execution. Another advantage of storing procedure in the database is that it will reduce the network traffic because most of the processing is done within the database itself.

In addition to these advantages, your application development efforts may also benefit. By consolidating the business rules within the database, they no longer need to be written into each application, saving time during application creation and simplifies the maintenance process. Procedural objects (procedures, functions and packages) can only be created in Oracle databases that have installed procedural option.

### C.14.1 Required System Privileges

In order to create a procedural object, you must have the CREATE PROCEDURE system privilege, which is part of the RESOURCE role. If the procedural object is in another user's schema, then you must have the CREATE ANY PROCEDURE system privilege.

### C.14.2 Executing Procedures

Once a procedural object has been created, it can be executed. When a procedural object is executed, it relies on the table privileges of its owner, not those of the user who is executing it. A user executing a procedure does not need to have been granted access to the tables that the procedure accesses. To allow other users to execute your procedural object, GRANT them EXECUTE privilege on that object as shown in the following example:

```
GRANT EXECUTE ON my_proc TO Alex;
```

The user Alex will now be able to execute the procedure named my\_proc—even if he does not have privileges on any of the tables that my\_proc uses. If you do not GRANT EXECUTE privileges to users, then they must have the EXECUTE ANY PROCEDURE system privilege to execute the procedure.

When executed, procedures usually have variables passed to them. For example, the RAISE\_APPLICATION\_ERROR procedure takes two input parameters: the error number (should be between -20001 and -20999) and error message to be displayed.

The syntax used to execute a procedure depends on the environment from which the procedure is being called. From within SQL\*Plus, a procedure can be executed using the EXECUTE command followed by the procedure name. Any arguments passed to the procedure must be enclosed in parentheses following the procedure name:

```
EXECUTE RAISE_APPLICATION_ERROR (-20002, 'Deletion of records only allowed by Sales users');
```

From within another procedure, function, package or trigger, the procedure can be called without the EXECUTE command:

### 30 ▲ Essentials of Database Management Systems

```
RAISE_APPLICATION_ERROR (-20002, 'Deletion of records only allowed by Sales users');
```

To execute a procedure owned by another user, you must create a synonym for that procedure or reference the owner's name during execution. For example consider executing the procedure `add_employee` owned by Mathews. For creating the synonym for the procedure, you can use the `CREATE SYNONYM` command. The owner of the synonym would then no longer need to refer to the procedure's owner in order to execute the procedure.

```
CREATE SYNONYM add_employee FOR Mathews.add_employee;  
EXECUTE add_employee ('ALEXIS LEON');
```

The second way of executing the procedure owned by another user is to reference the owner's name during execution:

```
EXECUTE Mathews.add_employee ('ALEXIS LEON');
```

When executing remote procedures, the name of a database link must be specified. The name of the database link must be specified after the procedure's name but before the variables:

```
EXECUTE add_employee@remote_link ('ALEXIS LEON');
```

The command above uses the `remote_link` database link to access the procedure called `add_employee` in a remote database. To make the location transparent to the user, a synonym may be created for the remote procedure:

```
CREATE SYNONYM add_employee FOR add_employee@remote_link;
```

Once the synonym is created, the user can refer to the remote procedure by using the name of the synonym. Oracle assumes that all remote procedure calls involve updates in the remote database. Therefore, you must have installed the distributed option in order to call remote procedures.

#### C.14.3 Required Table Privileges

Procedural objects may reference tables. In order for the objects to execute properly, the owner of the procedure, package, or function being executed must have privileges on the table it uses. The user who is executing the procedural object does not need privileges on underlying tables, he only needs to have `EXECUTE` privilege on the procedural object. The privileges needed for procedures, functions and packages cannot come from roles, they must be granted directly to the owner of the procedure, package or function.

#### C.14.4 Procedure vs. Functions

Unlike procedures, functions can return a value to the caller. This value is returned through the use of the `RETURN` keyword within the function.

#### C.14.5 Procedures vs. Packages

Packages are groups of procedures, functions, variables and SQL statements grouped together into a single unit. To execute a procedure within a package, you must first list the package name, then the procedure name:

```
EXECUTE employee_pack.add_employee ('ALEXIS LEON');
```

In the above example, the `add_employee` procedure in the `employee_pack` package was executed. Packages allow multiple procedures to use the same variables and cursors. Procedures within packages may be available to the public or they may be private, in which case they are only accessible via

commands from within the package, such as calls from other procedures. Procedures may also include commands that are to be executed each time the package is called, regardless of the procedure or function called within the package. Thus, they not only group procedures, but also give you the ability to execute commands that are not procedure-specific.

### C.14.6 Creating a Procedure

A procedure can be created using the CREATE PROCEDURE command:

```
CREATE [OR REPLACE] PROCEDURE [User.] procedure_name
[(argument [IN|OUT|IN OUT] data type
[,argument [IN|OUT|IN OUT] data type].....)]
{IS|AS} block;
```

Both the header and the body of the procedure are created by this command. The add\_employee procedure created by the command is shown below:

```
CREATE PROCEDURE add_employee (emp_name IN VARCHAR2)
AS
BEGIN
      INSERT INTO employee_table
      (name, age, salary)
      VALUE
      (emp_name, NULL, NULL);
END;
```

The add\_employee procedure shown above will accept the employee name as the input. It can be called from any application. It inserts a record into the employee\_table, with NULL values for age and salary columns.

If the procedure already exists you can replace it by the CREATE OR REPLACE PROCEDURE command. The benefit of using this command instead of dropping the procedure and re-creating it is that, the EXECUTE grants previously made on the procedure will remain untouched.

The IN qualifier is used for arguments for which values must be specified when calling the procedure. The OUT qualifier signifies that the procedure passes a value back to the caller through this argument. The IN OUT qualifier signifies that the argument is both an IN and an OUT: a value must be specified for this argument when the procedure is called and the procedure will return a value to the caller via this argument. The default qualifier is IN. The block refers to the PL/SQL block that the procedure will execute when called. The PL/SQL blocks within procedures can include DML statements but not DDL statement.

### C.14.7 Creating a Function

A procedure can be created using the CREATE FUNCTION command:

```
CREATE [OR REPLACE] FUNCTION [User.] function_name
[(argument IN data type
[,argument IN data type].....)]
RETURN data type
{IS|AS} block;
```

Both the header and the body of the procedure are created by this command. The only valid argument qualifier for functions is IN and may be omitted. The RETURN keyword specifies the data type of the

## 32 ▲ Essentials of Database Management Systems

function's return value. This can be any valid PL/SQL data type. Every function must have a RETURN clause, since the function must return a value to the calling environment. The following example shows a function emp\_salary, which returns the salary of the employee whose name is given as the input.

```
CREATE FUNCTION emp_salary (emp_name IN VARCHAR2)
RETURN NUMBER
IS
salary NUMBER (10,2);
BEGIN
    SELECT emp_salary
    INTO salary
    FROM employee
    WHERE name = emp_name;
    RETURN(salary);
END;
```

If the function already exists you can replace it by the CREATE OR REPLACE FUNCTION command. The benefit of using this command instead of dropping the procedure and re-creating it is that, the EXECUTE grants previously made on the procedure will remain untouched.

If the function is to be created in a different account, then you must have CREATE ANY PROCEDURE system privilege. If no schema is specified then the function will be created in your schema. In order to create a function in your schema, you must have the CREATE PROCEDURE system privilege (part of the RESOURCE role). Having the privilege to create procedures gives you the privilege to create functions and packages as well.

### C.14.8 Referencing Remote Tables in Procedures

Remote tables can be accessed by the SQL statements in procedures. A remote table can be queried by using the datalink in the procedure as shown in the following example:

```
CREATE PROCEDURE add_employee (emp_name IN VARCHAR2)
AS
BEGIN
    INSERT INTO employee_table@remote_link
    (name, age, salary)
    VALUE
    (emp_name, NULL, NULL);
END;
```

The add\_employee procedure inserts a record into the employee\_table, in the database defined by the remote\_link database link. But data manipulation in remote databases will require that the distributed option be installed in your database.

### C.14.9 Customizing Error Conditions

As with triggers, you may establish different error conditions within procedural objects. For each of the error conditions you define, you may select an error message that appears when the error occurs. The error numbers and messages that are displayed to the user are set through the RAISE\_APPLICATION\_ERROR procedure. This procedure may be called from within any procedural object. The RAISE\_APPLICATION\_ERROR procedure takes two input parameters: the error number (should be between -20001 and -20999) and error message to be displayed. You get to assign both message number and the text that will be displayed to the user. This is a very powerful addition to the standard exceptions that are available in the PL/SQL.

The following example shows a function `emp_salary`, which returns the salary of the employee whose name is given as the input. An additional section titled `EXCEPTIONS` tells Oracle how to handle nonstandard processing. In this example the `NO_DATA_FOUND` exception's standard message is overridden via the `RAISE_APPLICATION_ERROR` procedure.

```
CREATE FUNCTION emp_salary (emp_name IN VARCHAR2)
RETURN NUMBER
IS
salary NUMBER (10,2);
BEGIN
    SELECT emp_salary
    INTO salary
    FROM employee
    WHERE name = emp_name;
    RETURN(salary);
EXCEPTIONS
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR (-20100, 'No such employee exist.');
```

`END;`

In the above example, the `NO_DATA_FOUND` exception was used. If you wish to define custom exceptions, you need to name the exception in a `DECLARE` section of the procedure. The `DECLARE` section preceded the `BEGIN` command. As shown in the following example, this section should include entries for each of the custom exceptions you have defined, listed as type `'EXCEPTION'`:

```
DECLARE
custom_error EXCEPTION;
```

If you are using the exceptions already defined within PL/SQL, then you do not have to list them in the `DECLARE` section of the procedural object. In the `EXCEPTIONS` section of the procedural object's code, you tell the database how to handle the exceptions. It begins with the keyword `EXCEPTIONS`, followed by a `WHEN` clause for each exceptions. Each exception can call the `RAISE_APPLICATION_ERROR` procedure. You can use the `WHEN OTHERS` clause to handle all non-specified exceptions. The use of `RAISE_APPLICATION_ERROR` gives you great flexibility in managing the error conditions that may be encountered within your procedural objects.

### C.14.10 Creating a Package

When creating a package, the package specification and the package body are created separately. Thus there are two commands: `CREATE PACKAGE` and `CREATE PACKAGE BODY`. Both the commands require that you have the `CREATE PROCEDURE` system privilege. If the package is created in a schema that is not your own, then you must have the `CREATE ANY PROCEDURE` system privilege. The syntax for creating a package is:

```
CREATE [OR REPLACE] PACKAGE [User.] package_name
{IS|AS}
PL/SQL package specification;
```

A package specification consists of the list of functions, procedures, variables, constants, cursors and exceptions that will be available to users of the package. An example is shown below which contains the `emp_salary` function and `add_employee` procedure seen earlier in this chapter.

### 34 ▲ Essentials of Database Management Systems

```
CREATE PACKAGE emp_pack
AS
FUNCTION emp_salary (emp_name IN VARCHAR2);
PROCEDURE add-employee (emp_name IN VARCHAR2);
END;
```

A package body contains the PL/SQL blocks and specifications for all the public objects listed in the package specification. The package body may include objects that are not listed in the package specification. Such objects are said to be private and are not available to the user of the package but can be called only by other objects within the same package body. A package body may also include code that is run every time the package is invoked, regardless of the part of the package that is executed. The syntax for the package body is:

```
CREATE [OR REPLACE] PACKAGE BODY [User.] packagebody_name
{IS|AS}
PL/SQL package body;
```

The name of the package body should be same as the name of the package. The package body for the above package will be:

```
CREATE PACKAGE BODY emp_pack
AS
FUNCTION emp_salary (emp_name IN VARCHAR2)
RETURN NUMBER
IS
salary NUMBER (10,2);
BEGIN
SELECT emp_salary
INTO salary
FROM employee
WHERE name = emp_name;
RETURN(salary);
EXCEPTIONS
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR (-20100, 'No such employee exists.');
```

```
END emp_salary;
PROCEDURE add_employee (emp_name IN VARCHAR2)
AS
BEGIN
INSERT INTO employee_table
(name, age, salary)
VALUE
(emp_name, NULL, NULL);
END add_employee;
END emp_pack;
```

Modifying the END clauses by appending the names of the procedural object makes it easier to coordinate the logic with your code and improves readability.

### C.14.11 Initializing Packages

Package specifications may include code that is to be run the first time a user executes a package. In the following example the emp\_pack is modified to include an SQL statement that records the current user's username and the timestamp for the start of the package execution. Two new variables need to be declared in the package body to record these values. Since the variables are declared inside the package body they are not available to the public. Within the package body, they are separated from the procedures and functions. The package initialization code is shown in bold:

```

CREATE OR REPLACE PACKAGE BODY emp_pack
AS
User_name VARCHAR2;
E_Date DATE;
FUNCTION emp_salary (emp_name IN VARCHAR2)
RETURN NUMBER
IS
salary NUMBER (10,2);
BEGIN
    SELECT emp_salary
    INTO salary
    FROM employee
    WHERE name = emp_name;
    RETURN(salary);
    EXCEPTIONS
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR (-20100,
        'No such employee exists.');
```

```

END emp_salary;
PROCEDURE add_employee (emp_name IN VARCHAR2)
AS
BEGIN
    INSERT INTO employee_table
    (name, age, salary)
    VALUE
    (emp_name, NULL, NULL);
END add_employee;
BEGIN
SELECT User, SysDate
INTO User_name, E_Date
FROM DUAL;
END emp_pack;
```

The code that is to be run every time the package is executed is stored in its own PL/SQL block, at the bottom of the package body. It does not have its own END clause, it uses the package body's END clause. Every time the emp\_pack is executed, the User\_name and E-date variables are populated by the query. These two variables can then be used by the functions and procedures within the package.

To execute a procedure or function that is within the package, specify both the package name and the procedure/function name in the EXECUTE command as shown in the following example:

```
EXECUTE emp_pack.add_employee ('ALEXIS LEON');
```

### C.14.12 Viewing Source code for existing procedural objects

The source code for existing procedures, functions, packages and package bodies can be queried from the following data dictionary views:

- USER\_SOURCE - For procedural objects owned by the user
- ALL\_SOURCE - For procedural objects owned by the user or to which the user has been granted access
- DBA\_SOURCE - For procedural objects in the database

For example, to view the code of the procedure add\_employee, you can query the USER\_SOURCE as follows:

```
SELECT Text FROM USER_SOURCE
WHERE Name = 'add_employee' AND Type = 'PROCEDURE'
ORDER BY Line
```

The USER\_SOURCE view contains one record for each line of the procedure. The sequence of lines is maintained by the **Line** column and so the Line column should be used in the ORDER BY clause. Valid values for the Type column are PROCEDURE, FUNCTION, PACKAGE and PACKAGE BODY. The result of the query will be as shown in Figure 4.

Text
PROCEDURE add_employee (emp_name IN VARCHAR2)
AS
BEGIN
INSERT INTO employee_table
(name, age, salary)
VALUE
(emp_name, NULL, NULL);
END;

Figure 4 Viewing Source Code of Existing Procedures

### C.14.13 Compiling Procedures, Functions and Packages

Oracle compiles procedural objects when they are created. However, these may become invalid if the database objects they reference change. The next time the procedural objects are executed they will be recompiled by the database.

You can avoid this run-time compiling (and the resulting performance degradation) by explicitly recompiling the procedural objects. To recompile a procedure, use ALTER PROCEDURE command. The COMPILE clause is the only valid option for this command.

```
ALTER PROCEDURE add_employee COMPILE;
```

In order to use this command you must either own the procedure or have ALTER ANY PROCEDURE system privilege. The functions also can be recompiled in the same way using the command ALTER FUNCTION with the COMPILE option.

When recompiling packages, you may either recompile both the package specification and body or just the package body. The default is both. You cannot use the ALTER FUNCTION/PROCEDURE commands to recompile the functions/procedures within a package. The syntax for compiling the package is:

```
ALTER PACKAGE [User.] package_name
COMPILE [PACKAGE | BODY];
```

In order to use this command, you must either own the package or have ALTER ANY PROCEDURE system privilege. If the COMPILE clause is used without either PACKAGE or BODY both will be compiled.

## C.15 SUMMARY

PL/SQL is Oracle's procedural language extension to Oracle SQL. PL/SQL is a block-structured language that enables developers code procedures, functions and unnamed blocks that combine SQL with procedural statements. Related procedures, functions, cursors and variable declarations can be grouped and stored together in a package. Parameters can be passed to these blocks, enabling procedures and functions to accept and return values. PL/SQL fully supports ANSI/ISO SQL data manipulation language commands and SQL data types. Complex data is easily handled with PL/SQL's Boolean and composite data types. With blocks, you can group logically related declarations and statements. The declarations are local in state and cease to exist when the block completes. PL/SQL blocks can appear anywhere that SQL blocks can appear.

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 or to audit changes to data. The code within a trigger, called the trigger body, is made up of PL/SQL blocks.

Sophisticated business rules and application logic can be stored as **procedures** within Oracle. Stored procedures—groups of SQL and PL/SQL statements—allow you to move code that enforces business rules from your application to your database. As a result, the code will be stored once for all applications to use. Because Oracle supports stored procedures, the code within your application should become more consistent and easier to maintain.

You can group **procedures** and other PL/SQL commands into **packages**. There will be performance gains when using procedures. This is because of many reasons: One, the processing of complex business rules may be performed within the database and therefore by the server. In client/server applications, shifting complex processing from the application (on the client) to the database (on the server) may dramatically improve the performance. Second, since the procedural code is stored within the database and is fairly static, you may also benefit from the reuse of same queries within the database. Another advantage of storing procedure in the database is that it will reduce the network traffic because most of the processing is done within the database itself. In addition to these advantages, the application development efforts will also benefit from the use of procedures and packages. By consolidating the business rules within the database, they no longer need to be written into each application, saving time during application creation and simplifies the maintenance process.



## SELECTED BIBLIOGRAPHY

1. Celko, J., *Instant SQL Programming*, Wrox Press Inc., 1995.
2. Celko, J., *Joe Celko's SQL for Smarties: Advanced SQL Programming (2<sup>nd</sup> Edition)*, Morgan Kaufmann Publishers, 1999.
3. Colburn, R., *Using SQL*, QUE, 1999.
4. Couchman, J., and O'Hearn, S., *Oracle Certified Professional Developer PL/SQL Program Units Exam Guide*, McGraw-Hill, 2001.
5. Date, C. J. and Darwen, H., *A Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL*, Addison-Wesley Publishing Company, 1997.
6. Date, C. J., *A Guide to the SQL Standard: A User's Guide to the Standard Relational Language SQL (4<sup>th</sup> Edition)*, Addison-Wesley Publishing Company, 1997.
7. Feuerstein, S., *Oracle PL/SQL Programming: Guide to Oracle8i Features*, O'Reilly & Associates, 1999.
8. Pinnacle Software Solutions, Inc., *Oracle 8i: PL/SQL*, Pinnacle Software Solutions, Inc., 2001.
9. Pribyl, B., and Feuerstein, S., *Learning Oracle PL/SQL*, O'Reilly & Associates, 2001.

**38 ▲ Essentials of Database Management Systems**

10. Trezzo, J., *Oracle PL/SQL Tips and Techniques*, McGraw-Hill Professional Publishing, 1999.
11. Urman, S., and Rinaldi, W., *Oracle8 PL/SQL Programming*, McGraw-Hill Professional Publishing, 1997.
12. Urman, S., *Oracle Advanced PL/SQL Programming*, McGraw-Hill Professional Publishing, 2000.