PL/SQL

This section will provide a basic understanding of PL/SQL. This document will briefly cover the main concepts behind PL/SQL and provide brief examples illustrating the important facets of the language. Most of the information contained in this section is DIRECTLY extracted from ``PL/SQL User's Guide and Reference'' and all credit should be given to ORACLE. If you require more detailed information than provided in this section, consult the above stated manual.

PL/SQL is Oracle's procedural language extension to SQL, the relational database language. PL/SQL fully integrates modern software engineering features such as data encapsulation, information hiding, overloading, and exception handling, and so brings state-of-the-art programming to the ORACLE Server and a variety of ORACLE tools.

Overview of PL/SQL

With PL/SQL, you can use SQL statements to manipulate ORACLE data and flow-of-control statements to process the data. Moreover, you can declare constants and variables, define subprograms (procedures and functions), and trap runtime errors. Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

PL/SQL is a block-structured language. That is, the basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or subproblem to be solved.

A block (or sub-block) lets you group logically related declarations and statements. That way you can place declarations close to where they are used. The declarations are local to the block and cease to exist when the block completes.

```
[DECLARE
  -- declarations]
BEGIN
  -- statements
[EXCEPTION
  -- handlers]
END;
```

FUNDAMENTALS of PL/SQL

Lexical Units

PL/SQL is not case-sensitive, so lower-case letters are equivalent to corresponding upper-case letters except within string and character literals. A line of PL/SQL text contains groups of characters known as lexical units, which can be classified as follows:

•delimiters (simple and compound symbols)

•identifiers, which include reserved words

•literals

•comments

A delimiter is a simple or compound symbol that has a special meaning to PL/SQL. For example, you use delimiters to represent arithmetic operations such as addition and subtraction.

You use identifiers to name PL/SQL program objects and units, which include constants, variables, exceptions, cursors, subprograms, and packages. Some identifiers called RESERVED WORDS, have a special syntactic meaning to PL/SQL and so cannot be redefined. For flexibility, PL/SQL lets you enclose identifiers within double quotes. Quoted identifiers are seldom needed, but occasionally they can be useful.

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier.

•Two kinds of numeric literals can be used in arithmetic expressions: integers and reals.

•String literal is a sequence of zero or more characters enclosed by single quotes. All string literals except the null string (`') belong to type CHAR. PL/SQL is case-sensitive within string literals.

•Boolean literals are the predefined values TRUE and FALSE and the non-value NULL (which stands for a missing, unknown, or inapplicable value). Keep in mind that Boolean literals are not strings.

The PL/SQL compiler ignores comments but you should not. Adding comments to your program promotes readability and aids understanding. PL/SQL supports two comment styles: single-line and multiline. Single-line comments begin with a double hyphen (--) anywhere on a line and extend to the end of the line. Multiline comments begin with a slash-asterisk (/*), end with an asterisk-slash (*/), and can span multiple lines. You cannot nest comments.

Datatypes

Every constant and variable has a datatype, which specifies a storage format, constraints, and valid range of values. PL/SQL provides a variety of predefined scalar and composite datatypes. A scalar type has no internal components. A composite type has internal components that can be manipulated individually. PL/SQL Datatypes are similar to SQL's Datatypes but some of the common datatypes are discussed again. For more information on the PL/SQL Datatypes see Chapter 2 of ``PL/SQL User's Guide and Reference.''

•(NUMBER) You use the NUMBER datatype to store fixed or floating point numbers of virtually any size. You can specify precision, which is the total number of digits, and scale, which determines where rounding occurs.

```
NUMBER[(precision, scale)]
```

You cannot use constants or variables to specify precision and scale; you must use an integer literals.

•(CHAR) You use the CHAR datatype to store fixed-length character data. The CHAR datatype takes an optional parameter that lets you specify a maximum length up to 32767 bytes.

```
CHAR[(maximum_length)]
```

You cannot use a constant or variable to specify the maximum length; you must use an integer literal. If you do not specify the maximum length, it defaults to 1.

•(VARCHAR2) You use the VARCHAR2 datatype to store variable-length character data. The VARCHAR2 datatype takes a required parameter that lets you specify a maximum length up to 32767 bytes.

```
VARCHAR2(maximum_length)
```

You cannot use a constant or variable to specify the maximum length; you must use an integer literal.

•(BOOLEAN) You use the BOOLEAN datatype to store the values TRUE and FALSE and the non-value NULL. Recall that NULL stands for a missing, unknown, or inapplicable value. The BOOLEAN datatype takes no parameters.

•(DATE) You use the DATE datatype to store fixed-length date values. The DATE datatype takes no parameters. Valid dates for DATE variables include January 1, 4712 BC to December 31, 4712 AD. When stored in the database column, date values include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight.

Datatype Conversion

Sometimes it is necessary to convert a value from one datatype to another. PL/SQL supports both explicit and implicit (automatic) datatype conversions.

To specify conversions explicitly, you use built-in functions that convert values from one datatype to another. PL/SQL conversion functions are similar to those in SQL. For more information on conversion functions see Chapter 2 of ``PL/SQL User's Guide and Reference.''

When it makes sense, PL/SQL can convert the datatype of a value implicitly. This allows you to use literals, variables, and parameters of one type where another type is expected. If PL/SQL cannot determine which implicit conversion is needed, you get a compilation error. It is your responsibility to ensure that values are convertible. For instance, PL/SQL can convert the CHAR value '02-JUN-92' to a DATE value, but PL/SQL cannot convert the CHAR value 'YESTERDAY' to a DATE value.

Declarations

Your program stores values in variables and constants. As the program executes, the values of variables can change, but the values of constants cannot.

You can declare variables and constants in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that the value can be referenced. They can also assign an initial value and specify the NOT NULL constraint.

```
birthdate    DATE;
emp_count    SMALLINT := 0;
acct_id      VARCHAR2 (5) NOT NULL := 'AP001';
```

The first declaration names a variable of type DATE. The second declaration names a variable of type SMALLINT and uses the assignment operator (:=) to assign an initial value of zero to the variable. The third declaration names a variable of type VARCHAR2, specifies the NOT NULL constraint, and assigns an initial value of 'AP001' to the variable.

In constant declarations, the reserved word CONSTANT must precede the type specifier.

```
credit_limit CONSTANT REAL := 5000.00;
```

•Using DEFAULT. If you prefer, you can use the reserved word DEFAULT instead of the assignment operator to initialize variables and constants. You can also use DEFAULT to initialize subprogram parameters, cursor parameters, and fields in a user-defined record.

```
tax_year  SMALLINT DEFAULT 92;
valid     BOOLEAN DEFAULT FALSE;
```

•Using %TYPE. The %TYPE attribute provides the datatype of a variable, constant, or database column. Variables and constants declared using %TYPE are treated like those declared using a datatype name. For example in the declaration below, PL/SQL treats debit like a REAL(7,2) variable.

```
credit  REAL(7,2);
debit  credit%TYPE;
```

The %TYPE attribute is particularly useful when declaring variables that refer to database columns. You can reference a table and column, or you can reference an owner, table, and column.

```
my_dname  scott.dept.dname%TYPE;
```

Using %TYPE to declare my_dname has two advantages. First, you need not know the exact datatype of dname. Second, if the database definition of dname changes, the datatype of my_dname changes accordingly at run time.

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

```
DECLARE
  emp_rec   emp%ROWTYPE;
  CURSOR c1 is SELECT deptno, dname, loc FROM dept;
  dept_rec  c1%ROWTYPE
  ...
BEGIN
  SELECT * INTO emp_rec FROM emp WHERE ...
  ...
END;
```

Columns in a row and corresponding fields in a record have the same names and datatypes.

The column values returned by the SELECT statement are stored in fields. To reference a field, you use the dot notation.

```
IF emp_rec.deptno = 20 THEN ...
```

In addition, you can assign the value of an expression to a specific field.

```
emp_rec.ename := 'JOHNSON';
```

A %ROWTYPE declaration cannot include an initialization clause. However, there are two ways to assign values to all fields in a record at once. First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.

```
DECLARE
  dept_rec1    dept%ROWTYPE;
  dept_rec2    dept%ROWTYPE;
  CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
  dept_rec3    c1%ROWTYPE;
  dept_rec4    c1%ROWTYPE;
BEGIN
  ...
  dept_rec1 := dept_rec2;
  dept_rec4 := dept_rec3;
  ...
```

```
END;
```

But, because dept_rec2 is based on a table and dept_rec3 is based on a cursor, the following assignment is illegal:

```
dept_rec2 := dept_rec3;  -- illegal
```

Second, you can assign a list of column values to a record by using the SELECT and FETCH statement, as the example below shows. The column names must appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement.

```
DECLARE
  dept_rec     dept%ROWTYPE;
  ...
BEGIN
  SELECT deptno, dname, loc INTO dept_rec FROM dept
    WHERE deptno = 30;
  ...
END;
```

However, you cannot assign a list of column values to a record by using an assignment statement. Although you can retrieve entire records, you cannot insert them. For example, the following statement is illegal:

```
INSERT INTO dept VALUES (dept_rec);  -- illegal
```

Select-list items fetched by a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases. In the following example, you use an alias called wages:

```
DECLARE
  CURSOR my_cursor IS SELECT sal + NVL(comm,0) wages, ename
    FROM emp;
  my_rec       my_cursor%ROWTYPE;
BEGIN
  OPEN my_cursor;
  LOOP
    FETCH my_cursor INTO my_rec;
    EXIT WHEN my_cursor%NOTFOUND;
    IF my_rec.wages > 2000 THEN
      INSERT INTO temp VALUES (NULL, my_rec.wages, my_rec.ename);
    END IF;
  END LOOP;
  CLOSE my_cursor;
END;
```

PL/SQL does not allow forward references. You must declare a variable or constant before referencing it in other statements, including other declarative statements. However, PL/SQL does allow the forward declaration of subprograms.

Some languages allow you to declare a list of variables belonging to the same datatype. PL/SQL does not allow this. For example, the following declaration is illegal:

```
i, j, k      SMALLINT;  -- illegal
```

Naming Conventions

The same naming conventions apply to all PL/SQL program objects and units including constants, variables, cursors, exceptions, procedures, functions, and packages.

Within the same scope, all declared identifiers must be unique. So, even if their datatypes differ, variables and parameters cannot share the same name.

In potentially ambiguous SQL statements, the names of local variables and formal parameters take precedence over the names of database tables. For example, the following SELECT statement fails because PL/SQL assumes that emp refers to the formal parameter:

```
PROCEDURE calc_bonus (emp NUMBER, bonus OUT REAL) IS
  avg_sal REAL;
  ...
BEGIN
  SELECT AVG(sal) INTO avg_sal FROM emp WHERE ...
  ...
END;
```

In such cases, you can prefix the table name with a username, as follows:

```
PROECEDURE calc_bonus (emp NUMBER, bonus OUT REAL) IS
  avg_sal REAL;
  ...
BEGIN
  SELECT AVG(sal) INTO avg_sal FROM scott.emp WHERE ...
  ...
END;
```

The names of database columns take precedence over the names of local variables and formal parameters. For example, the following DELETE statement removes all employees from the emp table, not just KING, because ORACLE assumes that both enames in the WHERE clause refer to the database column:

```
DECLARE
  ename   CHAR(10) := 'KING';
BEGIN
  DELETE FROM emp WHERE ename = ename;
  ...
END;
```

In such cases, to avoid ambiguity, prefix the names of local variables and formal parameters with my_ as follows:

```
DECLARE
  my_ename   CHAR(10) := 'KING';
  ...
```

Or, use a block label to qualify references, as follows:

```
<<main>>
DECLARE
  ename   CHAR(10) := 'KING';
BEGIN
  DELETE FROM emp WHERE ename = main.ename;
  ...
END;
```

The next example shows that you can use a subprogram name to qualify references to local variables and formal parameters:

```
PROCEDURE calc_bonus (empno NUMBER, bonus OUT REAL) IS
  avg_sal REAL;
  name    CHAR(10);
  job     CHAR(15) := 'SALESMAN';
BEGIN
  SELECT AVG(sal) INTO avg_sal FROM emp
    WHERE job = calc_bonus.job;   -- refers to local variable
  SELECT ename INTO name FROM emp
    WHERE empno = calc_bonus.empno;  -- refers to parameter
  ...
END;
```

Scope and Visibility

References to an identifier are resolved according to its scope and visibility. The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier. An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.

For example, identifiers declared in a PL/SQL block are considered local to that block and global to all its sub-blocks. If a global identifier is redeclared in a sub-block, both identifiers remain in scope. Within the sub-block, however, only the local identifier is visible because you must use a qualified name to reference the global identifier.

Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks. The two objects represented by the identifier are distinct, and any change in one does not affect the other. Note that a block cannot reference identifiers declared in other blocks nested at the same level because those identifiers are neither local nor global to the block.

If you redeclare a identifier in a sub-block, you cannot reference the global identifier unless you use a qualified name. The qualifier can be the label of an enclosing block (or enclosing subprogram) as follows:

```
<<outer>>
DECLARE
  birthdate   DATE;
BEGIN
  ...
  DECLARE birthdate    DATE;
  BEGIN
    ...
    IF birthdate = outer.birthdate THEN
      ...
    END IF;
  END;
END outer;
```

Assignments

Variables and constants are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL. So, unless you expressly initialize a variable, its value is undefined, as the following example shows:

```
DELCARE
  count   INTEGER;
BEGIN
  count := count + 1;   -- assigns a null to count
END;
```

Therefore, never reference a variable before you assign it a value. Only the values TRUE and FALSE and the non-value NULL can be assigned to a Boolean variable. When applied to PL/SQL expressions, the relational operators return a Boolean value. So, the following assignment is legal:

```
DECLARE
  done    BOOLEAN;
BEGIN
  done := (count > 500);
  ...
END;
```

Alternatively, you can use the SELECT or FETCH statement to have ORACLE assign values to a variable.

```
SELECT ename, sal + comm INTO last_name, wages FROM emp
  WHERE empno = emp_id;
```

For each item in the SELECT list, there must be a corresponding variable in the INTO list. Also, each item must return a value that is implicitly convertible to the datatype of its corresponding variable.

Expressions and Comparisons

All expressions and comparisons are the same as those explained in the SQL Reference section. Some guidelines follow to help you prevent falling into common traps.

In general, do not compare real numbers for exact equality or inequality. It is also a good idea to use parentheses when doing comparisons.

Remember that a boolean variable is itself either true or false. So comparisons with the boolean values TRUE and FALSE are redundant. For example, assuming the variable done belongs to type BOOLEAN, the IF statement

```
IF done = TRUE THEN ...
```

can be simplified as follows:

```
IF done THEN ...
```

You can avoid some common mistakes by keeping in mind the following rules:

•comparisons involving nulls always yield NULL

•applying the logical operator NOT to a null yields NULL

•in conditional control statements, if the condition evaluates to NULL, its associated sequence of statements is not executed

Recall that applying the logical operator NULL to a null yields NULL.

If a null argument is passed to a function, a null is returned except in the following cases:

•(DECODE) The function DECODE compares its first argument to one or more search expressions, which are paired with result expressions. Any search or result expression can be null. If a search is successful, the corresponding result is returned. In the next example, if the value of rating is null, DECODE returns the value 1000:

```
credit_limit := (rating, NULL, 1000, 'B', 2000, 'A', 4000);
```

•(NVL) If its first argument is null, the function NVL returns the value of its second argument. In the following example, if hire_date is null, NVL returns the value of SYSDATE; otherwise, NVL returns the value of hire_date:

```
start_date := NVL(hire_date, SYSDATE);
```

•(REPLACE). If its second argument is null, the function REPLACE returns the value of its first argument whether the optional third argument is present or not. For instance, after the assignment:

```
new_string := REPLACE(old_string, NULL, my_string);
```

the values of old_string and new_string are the same.

Built-in Functions

PL/SQL provides many powerful functions to help you manipulate data. You can use them wherever expressions of the same type are allowed. Furthermore, you can nest them.

The built-in functions fall into the following categories:

• error-reporting functions

• number functions

• character functions

• conversion functions

• data functions

• miscellaneous functions

You can use all the built-in functions in SQL statements except the error-reporting functions SQLCODE and SQLERRM. In addition, you can use all the functions in procedural statements except the miscellaneous function DECODE.

Most functions are the same as those discussed in SQL Reference section except the ones that are discussed below.

Two functions, SQLCODE and SQLERRM, give you information about PL/SQL execution errors.

•(SQLCODE) function SQLCODE return NUMBER
Returns the number associated with the most recently raised exception. This function is meaningful only in an exception handler. Outside a handler, SQLCODE always returns zero.

For internal exceptions, SQLCODE returns the number of the associated ORACLE error. The NUMBER that SQLCODE returns is negative unless the ORACLE error is "no data found", in which case SQLCODE returns +100.

For user-defined exceptions, SQLCODE returns +1 unless you used the pragma EXCEPTION_INIT to associate the exception with an ORACLE error number, in which case SQLCODE returns that error number.

•**(SQLERRM)** function SQLERRM [(error_number NUMBER)] return CHAR
Returns the error message associated with the current value of SQLCODE. SQLERRM is meaningful only in an exception handler. Outside a handler, SQLERRM with no argument always returns the message ``ORA-0000:normal, successful completion.''

For internal exceptions, SQLERRM returns the message associated with the ORACLE error that occurred. The message begins with the ORACLE error code.

For user-defined exceptions, SQLERRM returns the message ``User-Defined Exception'' unless you used the pragma EXCEPTION_INIT to associate the exception with an ORACLE error number, in which case SQLERRM returns the corresponding error message.

You can pass the argument error_number to SQLERRM, in which case SQLERRM returns the message associated with error_number.

The following miscellaneous functions may be of use to you in PL/SQL coding.

•**(UID)** function UID return NUMBER
Returns the unique identification number assigned to the current ORACLE user. UID takes no arguments.

•**(USER)** function USER return VARCHAR2
Returns the username of the current ORACLE user. USER takes no arguments.

•**(USERENV)** function USERENV (str VARCHAR2) return VARCHAR2
Returns information about the current session. You can use the information to write an application audit trail table or to determine the language and character set are in use.\\

The string str can have any of the following values:

•'**ENTRYID**' returns an auditing entry identifier

•'**LANGUAGE**' returns the language, territory, and database character set in use

•'**SESSIONID**' returns the auditing session identifier

•'**TERMINAL**' returns the operating system identifier for the session terminal

You cannot specify the 'ENTRYID' or 'SESSIONID' option in SQL statements that access a remote database.

PL/SQL Tables

PL/SQL provides two composite datatypes: TABLE and RECORD. Objects of type TABLE are called PL/SQL tables, which are modeled as (but not the same as) database tables. PL/SQL tables use a primary key to give you array-like access to rows.

Like the size of a database table, the size of a PL/SQL table is unconstrained. That is, the number of rows in a PL/SQL table can increase dynamically. The PL/SQL table grows as new rows are added.

PL/SQL tables can have one column and a primary key, neither of which can be named. The column can belong to any scalar type, but the primary key must belong to type BINARY_INTEGER.

PL/SQL tables must be declared in two steps. First, you define a TABLE type, then declare PL/SQL tables of that type. You can declare TABLE types in the declarative part of any block, subprogram, or package using the syntax:

```
TYPE type_name IS TABLE OF
  { column_type | variable%TYPE | table.column%TYPE } [NOT NULL]
  INDEX BY BINARY_INTEGER;
```

where type_name is a type specifier used in subsequent declarations of PL/SQL tables and column_type is any scalar (not composite) datatype such as CHAR, DATE, or NUMBER. You can use the %TYPE attribute to specify a column datatype.
In the following example, you declare a TABLE type called EnameTabTyp:

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

Once you define type EnameTabTyp, you can declare PL/SQL tables of that type, as follows:

```
ename_tab    EnameTabTyp
```

The identifier ename_tab represents an entire PL/SQL table.

PL/SQL table is unconstrained because its primary key can assume any value in the range of values defined for BINARY_INTEGER. As a result, you cannot initialize a PL/SQL table in its declaration. For example, the following declaration is illegal:

```
ename_tab    EnameTabTyp := ('CASEY', 'STUART', 'CHU');
```

To reference rows in a PL/SQL table, you specify a primary key value using the array-like syntax

```
plsql_table_name(primary_key_value)
```

where primary_key_value belongs to type BINARY_INTEGER. The magnitude range of a BINARY_INTEGER value is -2**31-1 ... 2**31-1. For example, you reference the third row in PL/SQL table ename_tab as follows:

```
ename_tab(3) ...
```

You can assign the value of a PL/SQL expression to a specific row using the following syntax:

```
plsql_table_name(primary_key_value) := plsql_expression;
```

In the example below, you use a cursor FOR loop to load two PL/SQL tables. A cursor FOR loop implicitly declares its loop index as a record, opens the cursor associated with a given query, repeatedly fetches rows of values from the cursor into fields in the record, then closes the cursor.

```
DECLARE
  TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
    INDEX BY BINARY_INTEGER;
  ename_tab    EnameTabTyp;
  sal_tab      SalTabTyp;
```

```
  i          BINARY_INTEGER := 0;
  ...
BEGIN
  -- load employee names and salaries into PL/SQL tables
  For emprec IN (SELECT ename, sal FROM emp) LOOP
     i := i + 1;
     ename_tab(i) := emprec.ename;
     sal_tab(i) := emprec.sal;
  END LOOP;
  --process the tables
  process_sals(ename_tab, sal_tab);
  ...
END;
```

Until a row is assigned a value, it does not exist. If you try to reference an uninitialized row, PL/SQL raises the predefined exception NO_DATA_FOUND.

Remember that the size of PL/SQL table is unconstrained so, that if you want to maintain a row count, you must declare a variable for that purpose. A PL/SQL table can grow large, constrained only by available memory. When PL/SQL runs out of memory it raises the predefined exception STORAGE_ERROR.

You must use a loop to INSERT values from a PL/SQL table into a database column. Likewise, you must use a loop to FETCH values from a database column into a PL/SQL table. Therefore, you cannot reference PL/SQL tables in the INTO clause of a SELECT statement.

There is no straightforward way to delete rows from a PL/SQL table because the DELETE statement cannot specify PL/SQL tables. Setting a row to NULL does not work because the row remains and does not raise the exception NO_DATA_FOUND when referenced.

Although you cannot delete individual rows from a PL/SQL table, you can use a simple workaround to delete entire PL/SQL tables. First, declare another PL/SQL table of the same type and leave it empty. Later, when you want to delete the original PL/SQL tables, simply assign the empty table to them.

User-defined Records

You can use the %ROWTYPE attribute to declare a record that represents a row in a table or a row fetched by a cursor. However, you cannot specify the datatypes of fields in the record or define fields of your own. The composite datatype RECORD lifts those restrictions.

As you might expect, objects of type RECORD are called records. Unlike PL/SQL tables, records have uniquely named fields, which can belong to different datatypes. For example, suppose you have different kinds of data about an employee such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such fields as the name, salary, and hire date of an employee would let you treat the data as a logical unit.

Like PL/SQL tables, records must be declared in two steps. First, you define a RECORD type, then declare user-defined records of that type.

You can declare RECORD types in the declarative part of any block, subprogram, or package using the syntax
7

```
TYPE type_name IS RECORD
  (field_name1 {field_type | variable%TYPE | table.column%TYPE |
       table%ROWTYPE} [NOT NULL],
  (field_name2 {field_type | variable%TYPE | table.column%TYPE |
       table%ROWTYPE} [NOT NULL],
... );
```

where type_name is a type specifier used in subsequent declarations of records and field_type is any datatype including RECORD and TABLE. You can use the %TYPE or %ROWTYPE attribute to specify a field datatype. In the following example, you declare a RECORD type named DeptRecTyp:

```
DECLARE
   TYPE DeptRecTyp is RECORD
    (deptno     NUMBER(2) NOT NULL := 20,
     dname      dept.dname%TYPE,
     loc        dept.dname%TYPE);
   ...
```

Once you define type DeptRecTyp, you can declare records of that type, as follows:

```
dept_rec      DeptRecTyp;
```

The identifier dept_rec represents an entire record.

To reference individual fields in a record, you use the dot notation and the following syntax:

```
record_name.field_name
```

You can assign the value of a PL/SQL expression to a specific field by using the following syntax:

```
record_name.field_name := plsql_expression;
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once. This can be done in two ways. First, you can assign one record to another if they belong to the same datatype. Second, you can assign a list of column values to a record by using the SELECT or FETCH statement, as the example below shows. Just make sure the column names appear in the same order as the fields in your record.

```
DECLARE
   TYPE DeptRecTyp IS RECORD
    (deptno     NUMBER(2) NOT NULL := 20,
     dname      dept.dname%TYPE,
     loc        dept.dname%TYPE);
   dept_rec      DeptRecTyp;
   ...
BEGIN
   SELECT deptno, dname, loc INTO dept_rec FROM dept
      WHERE deptno = 30;
   ...
END;
```

Even if their fields match exactly, records of different types cannot be assigned to each other. Furthermore, a user-defined record and a %ROWTYPE record always belong to different types.

You cannot assign a list of values to a record by using an assignment statement. So, the following syntax is illegal:

```
record_name := (value1, value2, value3, ...);  -- illegal
```

Also, records cannot be tested for equality or inequality. For instance, the following IF condition is illegal:

```
IF dept_rec1 = dept_rec2 THEN  -- illegal
  ...
END IF;
```

PL/SQL lets you declare and reference nested records. That is, a record can be the component of another record. You can assign one nested record to another if they belong to the same datatype. Such assignments are allowed even if the containing records belong to different datatypes, as follows:

```
DECLARE
   TYPE TimeTyp IS RECORD
     (minute     SMALLINT,
      hour       SMALLINT);
   TYPE MeetingTyp IS RECORD
     (day        DATE,
      time       TimeTyp,          -- nested record
      place      CHAR(20),
      purpose    CHAR(50));
   TYPE PartyTyp IS RECORD
     (date       DATE,
      time       TimeTyp,          -- nested record
      loc        CHAR(15));
   meeting       MeetingTyp;
   seminar       MeetingTyp;
   party         PartyTyp;
   ...
BEGIN
   meeting.day := '26-Jun-91';
   meeting.time.minute := 45;
   meeting.time.hour := 10;
   ...
   seminar.time := meeting.time;
   party.time := meeting.time;
END;
```

CONTROL STRUCTURES

According to the structure theorem, any computer program can be written using the basic control structures which can be combined in any way necessary to deal with a given problem.

The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE, FALSE, or NULL). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Conditional Control: IF Statements

Often, it is necessary to take alternative actions depending on circumstances. The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

The third form of IF statement uses the keyword ELSIF (NOT ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
  sequence_of_statements1;
ELSIF condition2 THEN
  sequence_of_statements2;
ELSE
  sequence_of_statements3;
```

```
END IF;
```

Iterative Control: LOOP and EXIT Statements

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
  sequence_of_statements3;
  ...
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use the EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

```
LOOP
  ...
  IF ... THEN
     ...
     EXIT;  -- exit loop immediately
  END IF;
END LOOP;
-- control resumes here
```

The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition evaluates to TRUE, the loop completes and control passes to the next statement after the loop.

```
LOOP
  FETCH c1 INTO ...
  EXIT WHEN c1%NOTFOUND;  -- exit loop if condition is true
  ...
END LOOP;
CLOSE c1;
```

Until the condition evaluates to TRUE, the loop cannot complete. So, statements within the loop must change the value of the condition.

Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<label_name>>
LOOP
  sequence_of_statements;
  ...
```

```
END LOOP [label_name];
```

Optionally, the label name can also appear at the end of the LOOP statement.

With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete, then use the label in an EXIT statement.

```
<<outer>>
LOOP
  ...
  LOOP
    ...
    EXIT outer WHEN ...  -- exit both loops
  END LOOP;
  ...
END LOOP outer;
```

WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
  sequence_of_statements;
  ...
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If the condition evaluates to TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the condition evaluates to FALSE or NULL, the loop is bypassed and control passes to the next statement. Since the condition is tested at the top of the loop, the sequence might execute zero times.

FOR-LOOP

Whereas the number of iteration through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.

```
FOR counter IN [REVERSE] lower_bound..upper_bound LOOP
  sequence_of_statements;
  ...
END LOOP;
```

The lower bound need not be 1. However, the loop counter increment (or decrement) must be 1.

PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

```
SELECT COUNT(empno) INTO emp_count FROM emp;
FOR i IN 1..emp_count LOOP
  ...
END LOOP;
```

The loop counter is defined only within the loop. You cannot reference it outside the loop. You need not explicitly declare the loop counter because it is implicitly declared as a local variable of type INTEGER.

The EXIT statement allows a FOR loop to complete prematurely. You can complete not only the current loop, but any enclosing loop.

Sequential Control: GOTO and NULL statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can make the meaning and action of conditional statements clear and so improve readability.

```
BEGIN
   ...
   GOTO insert_row;
   ...
   <<insert_row>>
   INSERT INTO emp VALUES ...
END;
```

A GOTO statement cannot branch into an IF statement, LOOP statement, or sub-block. A GOTO statement cannot branch from one IF statement clause to another. A GOTO statement cannot branch out of a subprogram. Finally, a GOTO statement cannot branch from an exception handler into the current block.

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. It can, however, improve readability. Also, the NULL statement is a handy way to create stubs when designing applications from the top down.

Interaction With ORACLE

SQL Support

By extending SQL, PL/SQL offers a unique combination of power and ease of use. You can manipulate ORACLE data flexibly and safely because PL/SQL supports all SQL data manipulation commands (except EXPLAIN PLAN), transaction control commands, functions, pseudocolumns, and operators. However, PL/SQL does not support data definition commands such as CREATE, session control commands such as SET ROLES, or the system control command ALTER SYSTEM.

Data Manipulation

To manipulate ORACLE data, you use the INSERT, UPDATE, DELETE, SELECT, and LOCK TABLE commands.

Transaction Control

ORACLE is transaction oriented; that is, ORACLE uses transactions to ensure data integrity. A transaction is a series of SQL data manipulation statements that does a logical unit of work. For example, two UPDATE statements might credit one bank account and debit another. At the same instant, ORACLE makes permanent or undoes all database changes made by a transaction. If your program fails in the middle of a transaction, ORACLE detects the error and rolls back the transaction. Hence, the database is restored to its former state automatically.

You use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION commands to control transactions. COMMIT makes permanent any database changes made during the current transaction. Until you commit your changes, other users cannot see them. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction. Used with ROLLBACK, undoes part of a transaction. SET TRANSACTION establishes a read-only transaction.

SQL Functions

PL/SQL lets you use all the SQL functions including group functions, which summarize entire columns of ORACLE data.

SQL Pseudocolumns

PL/SQL recognizes the following SQL pseudocolumns, which return specific data items: CURRVAL, LEVEL, NEXTVAL, ROWID, and ROWNUM.

For example, NEXTVAL returns the next value in a database sequence. They are called pseudocolumns because they are not actual columns in a table but behave like columns. For instance, you can reference pseudocolumns in SQL statements. Furthermore, you can select values from a pseudocolumn. However, you cannot insert values into, update values in, or delete values from a pseudocolumn. Assume that you have declared empno_seq as a database sequence, then the following statement inserts a new employee number into the emp table:

```
INSERT INTO emp VALUES (empno_seq.NEXTVAL, new_ename, ...);
```

A sequence is a database object that generates sequential numbers. When you create a sequence, you can specify its initial value and an increment. CURRVAL returns the current value in a specified sequence. Before you can reference CURRVAL in a session, you must use NEXTVAL to generate a number.

LEVEL is used with the SELECT CONNECT BY statement to organize rows from a database table into a tree structure. LEVEL returns the level number of a node in a tree structure. The root is level 1, children of the root are level 2 and so on. You specify the direction in which the query walks the tree (down from the root or up from the branches) with the PRIOR operator. In the START WITH clause, you specify a condition that identifies the root of the tree.

ROWID returns the rowid (binary address) of a row in a database table.

ROWNUM returns a number indicating the order in which a row was selected from a table. If a SELECT statement includes an ORDER BY clause, ROWNUMs are assigned to the retrieved rows before the sort is done.

Operators

PL/SQL lets you use all the SQL comparison, set, and row operators in SQL statements.

Cursor Management

PL/SQL uses two types of cursors: implicit and explicit. PL/SQL declares a cursor implicitly for all SQL data manipulation statements, including queries that return only one row. However, for queries that return more than one row, you must declare an explicit cursor or use a cursor FOR loop

Explicit Cursors

The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria. When a query returns multiple rows, you can explicitly define a cursor to process the rows. You define a cursor in a declarative part of a PL/SQL block, subprogram, or package by naming it and specifying a query. Then, you use three commands to control the cursor: OPEN, FETCH, and CLOSE.

Forward references are not allowed in PL/SQL. So, you must declare a cursor before referencing it in other statements. When you declare a cursor, you name it and associate it with a specific query. The cursor name is an undeclared identifier, not a PL/SQL variable; it is used only to reference a query.

Cursors can take parameters, as the example below shows. A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be IN parameter.

```
CURSOR c1 (median IN NUMBER) IS
   SELECT job, ename FROM emp WHERE sal > median;
```

To declare formal cursor parameters, you use the syntax:

```
CURSOR name [ (parameter [, parameter, ...]) ] IS
```

where parameter stands for the following syntax:

```
variable_name [IN] datatype [{:= | DEFAULT} value]
```

OPENing the cursor executes the query and identifies the active set, which consists of all rows that meet the query search criteria. For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows. Rows in the active set are not retrieved when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.

The FETCH statement retrieves the rows in the active set one at a time. Each time FETCH is executed, the cursor advances to the next row in the active set. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their datatypes must be compatible. Any variables in the WHERE clause of the query associated with the cursor are evaluated only when the cursor is OPENed. As the following example shows, the query can reference PL/SQL variables within its scope:

```
DECLARE
   my_sal    emp.sal%TYPE;
   my_job    emp.job%TYPE;
   factor    INTEGER := 2;
   cursor c1 IS
      SELECT factor*sal FROM emp WHERE job = my_job;
BEGIN
   ...
   OPEN c1;   -- here factor equals 2
   LOOP
     FETCH c1 INTO my_sal;
     EXIT WHEN c1%NOTFOUND;
     ...
     factor := factor + 1;   -- does not affect FETCH
   END LOOP;
   CLOSE c1;
END;
```

Explicit Cursor Attributes

Each cursor that you explicitly define has four attributes: %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. When appended to the cursor name, these attributes let you access useful information about the execution of a multirow query. You can use explicit cursor attributes in procedural statements but not in SQL statements.

•Using %NOTFOUND. When a cursor is OPENed, the rows that satisfy the associated query are identified and form the active set. Rows are FETCHed from the active set one at a time. If the last fetch returned a row, %NOTFOUND evaluates to FALSE. If the last fetch failed to return a row (because the active set was empty), %NOTFOUND evaluates to TRUE. FETCH is expected to fail eventually, so when that happens, no exception is raised.

Before the first fetch, %NOTFOUND evaluates to NULL. So, if FETCH never executes successfully, the loop is never exited unless your EXIT WHEN statement is as follows:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

You can open multiple cursors, then use %NOTFOUND to tell which cursors have rows left to fetch.

•Using %FOUND. %FOUND is the logical opposite of %NOTFOUND. After an explicit cursor is open but before the first fetch, %FOUND evaluates to NULL. Thereafter, it evaluates to TRUE if the last fetch returned a row or to FALSE if no row was returned.

•Using %ROWCOUNT. When you open its cursor, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT returns a zero. Thereafter, it returns the number of rows fetched so far. The number is incremented if the latest fetch returned a row.

•Using %ISOPEN. %ISOPEN evaluates to TRUE if its cursor is open; otherwise, %ISOPEN evaluates to FALSE.

Implicit Cursors

ORACLE implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. PL/SQL lets you refer to the most recent implicit cursor as the ``SQL" cursor. So, although you cannot use the OPEN, FETCH, and CLOSE statements to control an implicit cursor, you can still use cursor attributes to access information about the most recently executed SQL statement.

Implicit Cursor Attributes

The SQL cursor has four attributes: %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. When appended to the cursor name (SQL), these attributes let you access information about the execution of INSERT, UPDATE, DELETE, and SELECT INTO statements. You can use implicit cursor attributes in procedural statements but not in SQL statements.

•Using %NOTFOUND. The features of %NOTFOUND are similar to those of the explicit cursor attributes but you must bear in mind the following: if a SELECT INTO fails to return a row, the predefined exception NO_DATA_FOUND is raised whether you check %NOTFOUND on the next line or not. The check for %NOTFOUND on the next line would be useless because when NO_DATA_FOUND is raised, normal execution stops and control transfers to the exception-handling part of the block. In this situation %NOTFOUND is useful in the OTHERS exception handler. Instead of coding a NO_DATA_FOUND handler, you find out if that exception was raised by checking %NOTFOUND.

```
DECLARE
   my_sal     NUMBER(7,2);
   my_empno   NUMBER(4);
```

```
BEGIN
   ...
   SELECT sal INTO my_sal FROM emp WHERE empno = my_empno;
   -- might raise NO_DATA_FOUND
EXCEPTION
   WHEN OTHERS THEN
     IF SQL%NOTFOUND THEN  -- check for 'no data found'
        ...
     END IF;
   ...
END;
```

However, a SELECT INTO that calls a SQL group function never raises the exception NO_DATA_FOUND. That is because group functions such as AVG and SUM always return a value or a null.

•Using %FOUND, %ROWCOUNT and %ISOPEN. These attributes are similar in use to those of explicit cursor attributes.

Packaged Cursors

You can separate a cursor specification from its body for placement in a package by using the RETURN clause:

```
CREATE PACKAGE emp_actions AS
   /* Declare cursor specification */
   CURSOR c1 RETURN emp%ROWTYPE
   ...
END emp_action;

CREATE PACKAGE BODY emp_actions AS
   /* Define cursor body */
   CURSOR c1 RETURN emp%ROWTYPE
      SELECT * FROM emp WHERE sal > 3000;
   ...
END emp_actions;
```

This way, you can change the cursor body without changing the cursor specification. A cursor specification has no SELECT statement because the RETURN clause defines the datatype of the result value.

A cursor body must have a SELECT statement and the same RETURN clause as its corresponding cursor specification. Furthermore, the number and datatypes of select-list items in the SELECT statement must match the RETURN clause.

Cursor FOR Loops

You can use a cursor FOR loop to simplify coding. A cursor FOR loop implicitly declares its loop index as a record of type %ROWTYPE, opens a cursor, repeatedly fetches rows of values from the active set into fields in the record, then closes the cursor when all rows have been processed or when you exit the loop.

You can pass parameters to a cursor used in a cursor FOR loop. In the following example, you pass a department number. Then, you compute the total wages paid to employees in that department. Also, you determine how many employees have salaries higher than $2000 and how many have commissions larger than their salaries.

```
DECLARE
   CURSOR emp_cursor(dnum NUMBER) IS
     SELECT sal, comm FROM emp WHERE deptno = dnum;
   total_wages     NUMBER(11,2) := 0;
   high_paid       NUMBER(4) := 0;
   higher_comm     NUMBER(4) := 0;
BEGIN
    /* the number of iterations will equal the number of rows *
     * returned by emp_cursor                                 */
    FOR emp_record IN emp_cursor(20) LOOP
       emp_record.comm := NVL(emp_record.comm,0);
       total_wages := total_wages + emp_record.sal +
          emp_record.comm;
       IF emp_record.sal > 2000 THEN
          high_paid := high_paid + 1;
       END IF;
   END LOOP;
   INSERT INTO temp VALUES (high_paid, higher_comm, 'Total Wages: '
        || TO_CHAR(total_wages));
   COMMIT;
END;
```

Overriding Default Locking

By default ORACLE locks data structures for you automatically. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking.

•Using FOR UPDATE. When declaring a cursor that will be referenced in the WHERE CURRENT OF clause of an UPDATE or DELETE statement, you must use the FOR UPDATE clause to acquire exclusive row locks. If present, the FOR UPDATE clause must appear at the end of the cursor declaration, as the following example shows.

```
DECLARE
  CURSOR c1 IS SELECT empno, sal FROM emp
    WHERE job = 'SALESMAN' AND comm > sal FOR UPDATE;
```

The FOR UPDATE clause indicates that rows will be updated or deleted and locks all rows in the active set. All rows in the active set are locked when you OPEN the cursor. The rows are unlocked when you COMMIT the transaction. So, you cannot FETCH from a FOR UPDATE cursor after a COMMIT. When querying multiple tables, you can use the FOR UPDATE OF clause to confine row locking to particular tables.

```
DECLARE
  CURSOR c1 IS SELECT ename, dname FROM emp, dept
    WHERE emp.deptno = dept.deptno AND job = 'MANAGER'
    FOR UPDATE OF sal;
```

•Using a LOCK TABLE statement lets you lock entire database tables in a specified lock mode so that you can share or deny access to tables while maintaining their integrity. Table locks are released when your transaction issues a COMMIT or ROLLBACK.

```
LOCK TABLE emp IN ROW SHARE MODE NOWAIT;
```

Database Triggers

A database trigger is a stored PL/SQL program unit associated with a specific database table. ORACLE executes (fires) the database trigger automatically whenever a given SQL operation affects the table. So, unlike subprograms, which must be invoked explicitly, database triggers are invoked implicitly. Among other things, you can use database triggers to

•audit data modification

•log events transparently

•enforce complex business rules

•derive column values automatically

•implement complex security authorizations

•maintain replicate tables

You can associate up to 12 database triggers with a give table. A database trigger has three parts: a triggering event, an optional trigger constraint, and a trigger action. When the event occurs, the database trigger fires and an anonymous PL/SQL block performs the action. Database triggers fire with the privileges of the owner, not the current user. So, the owner must have appropriate access to all objects referenced by the trigger action.

The example below illustrates transparent event logging. The database trigger named reorder ensures that a part is reordered when its quantity on hand drops below the reorder point.

```
CREATE TRIGGER reorder
   /* triggering event */
   AFTER UPDATE OF qty_on_hand ON inventory  -- table
   FOR EACH ROW
      /* trigger constraint */
      WHEN (new.reorderable = 'T')
BEGIN
   /* trigger action */
   IF :new.qty_on_hand < :new.reorder_point THEN
     INSERT INTO pending_orders
        VALUES (:new.part_no, :new.reorder_qty, SYSDATE);
   END IF;
END;
```

The name in the ON clause identifies the database table associated with the database trigger. The triggering event specifies the SQL data manipulation statement that affects the table. In this case, the statement is UPDATE. If the trigger statement fails, it is rolled back. The keyword AFTER specifies that the database trigger fires after the update is done.

By default, a database trigger fires once per table. The FOR EACH ROW option specifies that the trigger fires once per row. For the trigger to fire, however, the Boolean expression in the WHEN clause must evaluate to TRUE.

The prefix :new is a correlation name that refers to the newly updated column value. Within a database trigger, you can reference :new and :old values of changing rows. Notice that the colon is not used in the WHEN clause. You can use the REFERENCING clause (not shown) to replace :new and :old with other correlation names.

Except for transaction control statements such as COMMIT and ROLLBACK, any SQL or procedural statement, including subprogram calls, can appear in the BEGIN...END block. A database trigger can also have DECLARE and EXCEPTION sections.

The next example shows that the trigger action can include calls to the built-in ORACLE procedure raise_application_error, which lets you issue user-defined error messages:

```
CREATE TRIGGER check_salary
   BEFORE INSERT OR UPDATE OF sal, job ON emp
   FOR EACH ROW
   WHEN (new.job != 'PRESIDENT')
DECLARE
   minsal   NUMBER;
   maxsal   NUMBER;
BEGIN
   /* Get salary range for a given job from table sals. */
   SELECT losal, hisal INTO minsal, maxsal FROM sals
      WHERE job = :new.job;
   /* If salary is out of range, increase is negative, *
    * or increase exceeds 10%, raise an exception.     */
   IF (:new.sal < minsal OR :new.sal > maxsal) THEN
      raise_application_error(-20225, 'Salary out of range');
   ELSIF (:new.sal < :old.sal) THEN
      raise_application_error(-20320, 'Negative increase');
   ELSIF (:new.sal > 1.1 * :old.sal) THEN
      raise_application_error(-20325, 'Increase exceeds 10%');
   END IF:
END;
```

More information on built-in procedures is provided later in this chapter. For a full discussion of database triggers, see ``ORACLE7 Server Application Developer's Guide''.

Error Handling

Overview

In PL/SQL a warning or error condition is called an exception. Exceptions can be internally defined (by the runtime system) or user-defined.

Examples of internally defined exceptions include division by zero and out of memory. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named insufficient_funds to flag an overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions must be given names.

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

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

Advantages of Exceptions

Using exceptions for error handling has several advantages. Without exception handling, every time you issue a command, you must check for execution errors. Exceptions also improve reliability. You need not worry about checking for an error at every point it might occur. Just add an exception handler to your PL/SQL block. If the exception is ever raised in that block (or any sub-block), you can be sure it will be handled.

Predefined Exceptions

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

PL/SQL declares predefined exceptions globally in package STANDARD, which defines the PL/SQL environment. So, you need not declare them yourself. You can write handlers for predefined exceptions using the names shown below:

Exception Name	ORACLE Error	SQLCODE Value
CURSOR_ALREADY OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
STORAGE_ERROR	ORA-06500	-6500
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

•**CURSOR_ALREADY_OPEN** is raised if you try to OPEN an already open cursor.

•**DUP_VAL_ON_INDEX** is raised if you try to store duplicate values in a database column that is constrained by a unique index.

•**INVALID_CURSOR** is raised if you try an illegal cursor operation. For example, if you try to CLOSE an unopened cursor.

•**INVALID_NUMBER** is raised in a SQL statement if the conversion of a character string to a number fails.

•**LOGIN_DENIED** is raised if you try logging on to ORACLE with an invalid username/password.

•**NO_DATA_FOUND** is raised if a SELECT INTO statement returns no rows or if you reference an uninitialized row in a PL/SQL table.

•**NOT_LOGGED_ON** is raised if your PL/SQL program issues a database call without being logged on to ORACLE.

•**PROGRAM_ERROR** is raised if PL/SQL has an internal problem.

•**STORAGE_ERROR** is raised if PL/SQL runs out of memory or if memory is corrupted.

•**TIMEOUT_ON_RESOURCE** is raised if a timeout occurs while ORACLE is waiting for a resource.

•**TOO_MANY_ROWS** is raised if a SELECT INTO statement returns more than one row.

•**VALUE_ERROR** is raised if an arithmetic, conversion, truncation, or constraint error occurs.

•**ZERO_DIVIDE** is raised if you try to divide a number by zero.

User-defined Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by RAISE statements. Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword EXCEPTION.

```
DECLARE
  past_due    EXCEPTION;
  acct_num    NUMBER(5);
BEGIN
```

Exceptions and variable declarations are similar. But remember, an exception is an error condition, not an object. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

•Using EXCEPTION_INIT. To handle unnamed internal exceptions, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A pragma is a compiler directive, which can be thought of as a parenthetical remark to the compiler.

In PL/SQL, the predefined pragma EXCEPTION_INIT tells the compiler to associate an exception name with an ORACLE error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package using the syntax

```
PRAGMA EXCEPTION_INIT(exception_name, ORACLE_error_number);
```

where exception_name is the name of a previously declared exception.

```
DECLARE
  insufficient_privileges    EXCEPTION;
  PRAGMA EXCEPTION_INIT(insufficient_privileges, -1031);
    ------------------------------------------------------
    -- ORACLE returns error number -1031 if, for example
    -- you try to UPDATE a table for which you have only
    -- SELECT privileges
    ------------------------------------------------------
BEGIN
  ...
EXCEPTION
  WHEN insufficient_privileges THEN
    -- handle the error
  ...
END;
```

•Using raise_application_error. A package named DBMS_STANDARD (part of the Procedural Database Extention) provides language facilities that help your application interact with ORACLE. This package includes a procedure named raise_application_error, which lets you issue user-defined error messages from a stored subprogram or database trigger. The calling syntax is

```
raise_application_error(error_number, error_message);
```

where error_number is a negative integer in the range -20000..-20999 and error_message is a character string up to 512 bytes in length.

An application can call raise_application_error only from an executing stored subprogram. When called, raise_application_error ends a subprogram, rolls back any database changes it made, and returns a user-defined error message to the application.

```
PROCEDURE raise_salary (emp_id NUMBER, increase NUMBER) IS
  current_salary      NUMBER;
BEGIN
  SELECT sal INTO current_salary FROM emp
    WHERE empno = emp_id;
  IF current_salary is NULL THEN
    raise_application_error(-20101, 'Salary is missing');
  ELSE
    UPDATE emp SET sal = current_salary + increase
      WHERE empno = emp_id;
  END IF;
END raise_salary;
```

The calling application gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS handler. Furthermore, it can use EXCEPTION_INIT to map specific error numbers returned by raise_application_error to exceptions of its own.

```
DECLARE
  ...
  null_salary   EXCEPTION;
  PRAGMA EXCEPTION_INIT(null_salary, -20101);
  ...
```

How Exceptions Are Raised

Internal exceptions are raised implicitly by the runtime system, as are user-defined exceptions that you have associated with an ORACLE error number using EXCEPTION_INIT. However, other user-defined exceptions must be raised explicitly by RAISE statements.

•Using RAISE statement. PL/SQL blocks and subprograms should RAISE an exception only when an error makes it undesirable or impossible to finish processing. You can code a RAISE statement for a given exception anywhere within the scope of that exception.

```
DECLARE
  out_of_stock     EXCEPTION;
  number_on_hand   NUMBER(4);
BEGIN
  ...
  IF number_on_hand < 1 THEN
    RAISE out_of_stock;
```

```
   END IF;
   ...
EXCEPTION
   WHEN out_of_stock THEN
     -- handle the error
END;
```

You can also raise a predefined exception explicitly:

```
RAISE INVALID_NUMBER;
```

That way, you can use an exception handler written for the predefined exception to process other errors.

Sometimes, you want to reraise an exception, that is, handle it locally, then pass it to an enclosing block. To reraise an exception, simply place a RAISE statement in the local handler, as shown below

```
DECLARE
   out_of_balance     EXCEPTION;
BEGIN
   ...
   ------------- beginning of sub-block ----------------------
   BEGIN
     ...
     IF ... THEN
     RAISE out_of_balance;  -- raise the exception
   END IF;
     ...
   EXCEPTION
       WHEN out_of_balance THEN
         -- handle the error
         RAISE;      -- reraise the current exception
   END;
   ------------- end of sub-block ----------------------------
EXCEPTION
   WHEN out_of_balance THEN
       - handle the error differently
   ...
END;
```

Omitting the exception name in a RAISE statement, which is allowed only in an exception handler, reraises the current exception.

Handling Raised Exceptions

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part and control does NOT return to where the exception was raised. In other words, you cannot resume processing where you left off.

The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions not named specifically. Use of the OTHERS handler guarantees that no exception will go unhandled.

If you want two or more exceptions to execute the same sequence of statements, list the exception names in the WHEN clause, separating them by the keyword OR. The keyword OTHERS cannot appear in the list of exception names; it must appear by itself.

```
...
EXCEPTION
   WHEN ... THEN
      - handle the error differently
   WHEN ... OR ... THEN
      - handle the error differently
   ...
   WHEN OTHERS THEN
      - handle the error differently
END;
```

• Using SQLCODE and SQLERRM. You cannot use SQLCODE and SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement.

```
DECLARE
   err_num    NUMBER;
   err_msg    CHAR(100);
BEGIN
   ...
   WHEN OTHERS THEN
     err_num := SQLCODE;
     err_msg := SUBSTR(SQLERRM, 1, 100);
     INSERT INTO errors VALUES (err_num, err_msg);
END;
```

The string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to err_msg. SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell you which internal exception was raised.

Subprograms

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprograms called procedures and functions. Generally, you use a procedure to perform an action and a function to compute a value.

Like unnamed or anonymous PL/SQL blocks, subprograms have a declarative part, an executable part, and an optional exception-handling part.

Procedures

A procedure is a subprogram that performs a specific action. You write procedures using the syntax

```
PROCEDURE name [ (parameter, [, parameter, ...]) ] IS
   [local declarations]
BEGIN
   executable statements
[EXCEPTION]
   exception-handlers]
END [name];
```

where parameter stands for the following syntax

```
var_name [IN | OUT | IN OUT] datatype [{:= | DEFAULT} value]
```

Unlike the datatype specifier in a variable declaration, the datatype specifier in a parameter declaration must be unconstrained.

```
PROCEDURE ... (name CHAR(20) ) IS  -- illegal; should be CHAR
```

The procedure specification begins with the keyword PROCEDURE and ends with the procedure name or a parameter list. The procedure body begins with the keyword IS and ends with the keyword END followed by an optional procedure name.

```
PROCEDURE raise_salary (emp_id INTEGER, increase REAL) IS
   current_salary  REAL;
   salary_missing  EXCEPTION;
BEGIN
   SELECT sal INTO current_salary FROM emp
      WHERE empno = emp_id;
   IF current_salary IS NULL THEN
     RAISE salary_missing;
   ESLE
     UPDATE emp SET sal = sal + increase
       WHERE empno = emp_id;
   END IF;
[EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO emp_audit VALUES (emp_id, 'No such number');
   WHEN salary_missing THEN
      INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
END raise_salary;
```

Functions

A function is a subprogram that computes a value. Functions and procedure are structured alike, except that functions have a RETURN clause. You write functions using the syntax

```
FUNCTION name [ (parameter, [, parameter, ...]) ] RETURN datatype IS
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception-handlers]
END [name];
```

where parameter stands for the following syntax

```
var_name [IN | OUT | IN OUT] datatype [{:= | DEFAULT} value]
```

The function body begins with the keyword IS and ends with the keyword RETURN clause, which specifies the datatype of the result value.

Calls to user-defined functions can appear in procedural statements, but not in SQL statements.

```
FUNCTION sal_ok (salary REAL, title REAL) RETURN BOOLEAN IS
   min_sal   REAL;
   max_sal   REAL;
BEGIN
   SELECT losal, hisal INTO min_sal, max_sal FROM sals
     WHERE job = title;
   RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;
```

RETURN Statement

The RETURN statement immediatedly completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. A subprogram can contain several RETURN statements, none of which need be the last lexical statement.

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

However, in functions, a RETURN statement must contain an expression, which is evaluated when the RETURN statement is reached. A function must contain at least one RETURN statement. Otherwise, PL/SQL raises the predefined exception PROGRAM_ERROR at run time.

Forward Declarations

PL/SQL requires that you declare an identifier before using it. Therefore, you must declare a subprogram before calling it. PL/SQL solves the problem of subprograms used before they are declared by providing a special subprogram declaration called forward declaration.

A forward declaration consists of a subprogram specification terminated by a semicolon.

```
DECLARE
   PROCEDURE calc_rating (...);  -- forward declaration
   /* Define subprogram in alphabetical order */
   PROCEDURE award_bonus (..) IS
   BEGIN
     calc_rating(..);
     ...
   END;

   PROCEDURE calc_rating (...) IS
   BEGIN
     ...
   END;
   ...
```

Although the formal parameter list appears in the forward declaration, it must also appear in the subprogram body. You can place the subprogram body anywhere after the forward declaration, but they must appear in the same block, subprogram, or package.

Packaged Subprograms

Forward declarations also let you group logically related subprograms in a package. The subprogram specifications go in the package specification, and the subprogram bodies go in the package body, where they are invisible to applications. Thus, packages allow you to hide implementation details.

Actual versus Formal Parameters

Subprograms pass information using parameters. The variables or expressions referenced in the parameter list of a subprogram call are actual parameters. The variables declared in a subprogram specification and referenced in the subprogram body are formal parameters.

The actual parameter and its corresponding formal parameter must belong to compatible datatypes.

When calling a subprogram, you can write the actual parameters using either positional or named notation. For example, the call to the procedure raise_salary can be made as follows:

```
raise_salary(emp, inc);
raise_salary(increase => inc, emp_id => emp)
raise_salary(emp, increase => inc)
```

The first procedure call uses positional notation, the second uses named notation, and the third uses mixed notation.

Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes, IN (the default), OUT, and IN OUT, can be used with any subprogram. However, avoid using the OUT and IN OUT modes with functions.

•an IN parameter lets you pass values to the subprogram being called. Inside the subprogram, an IN parameter acts like a constant. Therefore, it cannot be assigned a value. Unlike OUT and IN OUT parameters, an IN parameter can be initialized to default values.

•an OUT parameter lets you return values to the caller of a subprogram. Inside the subprogram, an OUT parameter acts like an uninitialized variable. Therefore, its value cannot be assigned to another variable or reassigned to itself.

The actual parameter that corresponds to an OUT formal parameter must be a variable; it cannot be a constant or expression.

An OUT actual parameter can (but need not) have a value before the subprogram is called. However, the value is lost when you call the subprogram.

Before exiting a subprogram, explicitly assign values to all OUT formal parameters. Otherwise, the values of corresponding actual parameters are indeterminate. If you exit successfully, PL/SQL assigns values to the actual parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

•an IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, an IN OUT parameter acts like an initialized variable.
The actual parameter that corresponds to an IN OUT formal parameter must be a variable; it cannot be a constant or expression.

Overloading

PL/SQL lets you overload subprogram names. That is, you can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

```
DECLARE
  TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
  TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
  hiredate_tab   DateTabTyp;
  sal_tab        RealTabTyp;
  ...
```

You might write the following procedures to initialize the PL/SQL tables named initialize for hiredate_tab and sal_tab.

```
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
   FOR i IN 1..n LOOP
     tab(i) := SYSDATE;
   END LOOP;
END initialize;

PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
BEGIN
   FOR i IN 1..n LOOP
     tab(i) := 0.0;
   END LOOP;
END initialize;
```

Because the processing in these two procedures is the same, it is logical to give them the same name. You can place the two overloaded initialize procedures in the same block, subprogram, or package. PL/SQL determines which of the two procedures is being called by checking their formal parameters.

You cannot overload the names of stand-alone subprograms. You cannot overload two subprograms if their formal parameters differ only in name or parameter mode. You cannot overload two subprograms if their formal parameters differ only in datatype and the different datatypes are in the same family (REAL and INTEGER). Finally, you cannot overload two functions that differ only in return type even if the types are in different families.

Stored Subprograms

Subprograms can be compiled separately and stored permanently in an ORACLE database, ready to be executed.

Stored subprograms offer higher productivity, better performance, memory savings, application integrity, and tighter security. Stored subprograms can help enforce data security. You can restrict users to specific database operations by granting access only through subprograms. For example you can grant users EXECUTE access to a stored procedure that updates the emp table, but not grant them access to the table itself. That way, users can call the procedure, but cannot arbitrarily manipulate table data.

You can call stored subprograms from a database trigger, another stored subprogram, an ORACLE Precompiler application, an OCI application, or an ORACLE tool such as SQL*Plus.

PACKAGES

A package is a database object that groups logically related PL/SQL types, objects, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The specification is the

interface to your application; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the specification.

Unlike subprograms, packages cannot be called, passed parameters, or nested. Still, the format of a package is similar to that of a subprogram:

```
PACKAGE name IS  -- specification (visible part)
   -- public type and object declarations
   -- subprogram specifications
END [name};

PACKAGE BODY name IS  -- body (hidden part)
  -- private type and object declarations
  -- subprogram bodies
[BEGIN
  -- initialization statements]
END [name];
```

Packages are created interactively with SQL*Plus using the CREATE PACKAGE and CREATE PACKAGE BODY commands. In the following example, a record type, a cursor, and two employment procedures are packaged:

```
CREATE PACKAGE emp_actions AS   -- specification
   TYPE EmpRecTyp is RECORD (emp_id INTEGER, salary REAL);
   CURSOR desc_salary (emp_id NUMBER) RETURN EmpRecTyp;

   PROCEDURE hire_employee
     (ename CHAR, job CHAR, mgr NUMBER, sal NUMBER,
     comm NUMBER, deptno NUMBER );
   PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE PACKAGE BODY emp_actions AS   -- body
   CURSOR desc_salary (emp_id NUMBER) RETURN EmpRecTyp IS
     SELECT empno, sal FROM emp ORDER BY sal DESC;

   PROCEDURE hire_employee
      (ename CHAR, job CHAR, mgr NUMBER, sal NUMBER,
       comm NUMBER, deptno NUMBER );
   BEGIN
      INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job, mgr,
          SYSDATE, sal, comm, deptno );
   END hire_employee;

   PROCEDURE fire_employee (emp_id NUMBER) IS
   BEGIN
      DELETE FROM emp WHERE empno = emp_id;
   END fire_employee;
END emp_actions;
```

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

The Package Specification

The package specification contains public declarations. The scope of these declarations is local to your database schema and global to the package. The specification lists the package resources available to applications. All information your application needs to use the resources is in the specification.

Only subprograms and cursors have an underlying implementation or definition. So, if a specification declares only types, constants, variables, and exceptions, the package body is unnecessary.

To reference the types, objects, and subprograms declared within a package specification, you use dot notation as follows:

```
package_name.type_name
package_name.object_name
package_name.subprogram_name
```

The Package Body

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

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

Following the declarative part of a package body is the optional initialization part, which typically holds statements that initialize some of the variables previously declared in the package.

The initialization part of a package plays a minor role because, unlike subprograms, a package cannot be called or passed parameters. As a result, the initialization part of a package is run only once, the first time you reference the package.

Guidelines

When writing packages, keep them as general as possible so they can be reused in future applications. Avoid writing packages that duplicate some feature already provided by ORACLE.

Package specifications reflect the design of your application. So, define them before the package bodies. Place in a specification only the types, objects, and subprograms that must be visible to users of the package.

To reduce the need for recompiling when code is changed, place as few items as possible in a package specification. Changes to a package body do not require ORACLE to recompile dependent procedures. However, changes to a package specification require ORACLE to recompile every stored subprogram that references the package.