PL/SQL for Oracle 8/8i

PL/SQL

PL/SQL

- **Block Structure**
- **Datatypes**
- **Declarations**
- **Scope & Visibility**
- **Control Constructs**
- **Cursors**
- **Error Handling**
- **Sub Programs**
- **Overloading Subprograms**
- **Stored Procedures and Functions**
- **Packages**

PL/SQL

Object Types

External Procedures

Triggers

New PL/SQL Features in Oracle 8i

What is PL/SQL?

PL/SQL is Oracle's Procedural Language extension to SQL.

The PL/SQL language includes object oriented programming techniques such as encapsulation, function overloading, information hiding (all but inheritance) - Hence bridging the gap between database technology and other programming languages.

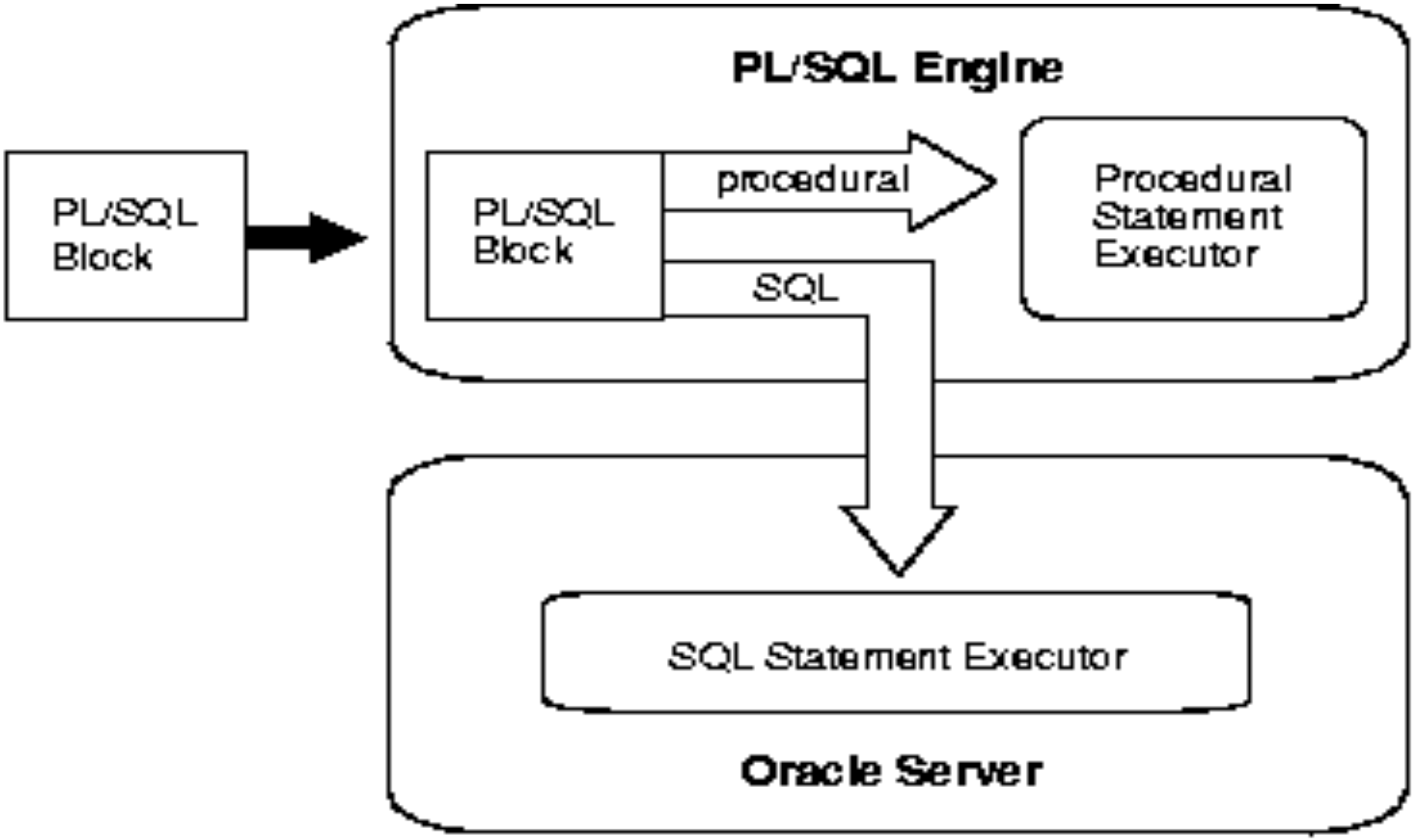
PL/SQL permits the use of all SQL data manipulation statements as well as transaction processing statements.

Basic PL/SQL Block Structure

PL/SQL is structured into blocks and can use conditional statements, loops and branches to control program flow.

```
DECLARE
    /* Definition of any variables or objects that are used
    within the declared block. */
BEGIN
    -- Statements that make up the block.
EXCEPTION
    -- All exception handlers.
END
    -- End of block marker.
```

PL/SQL Execution



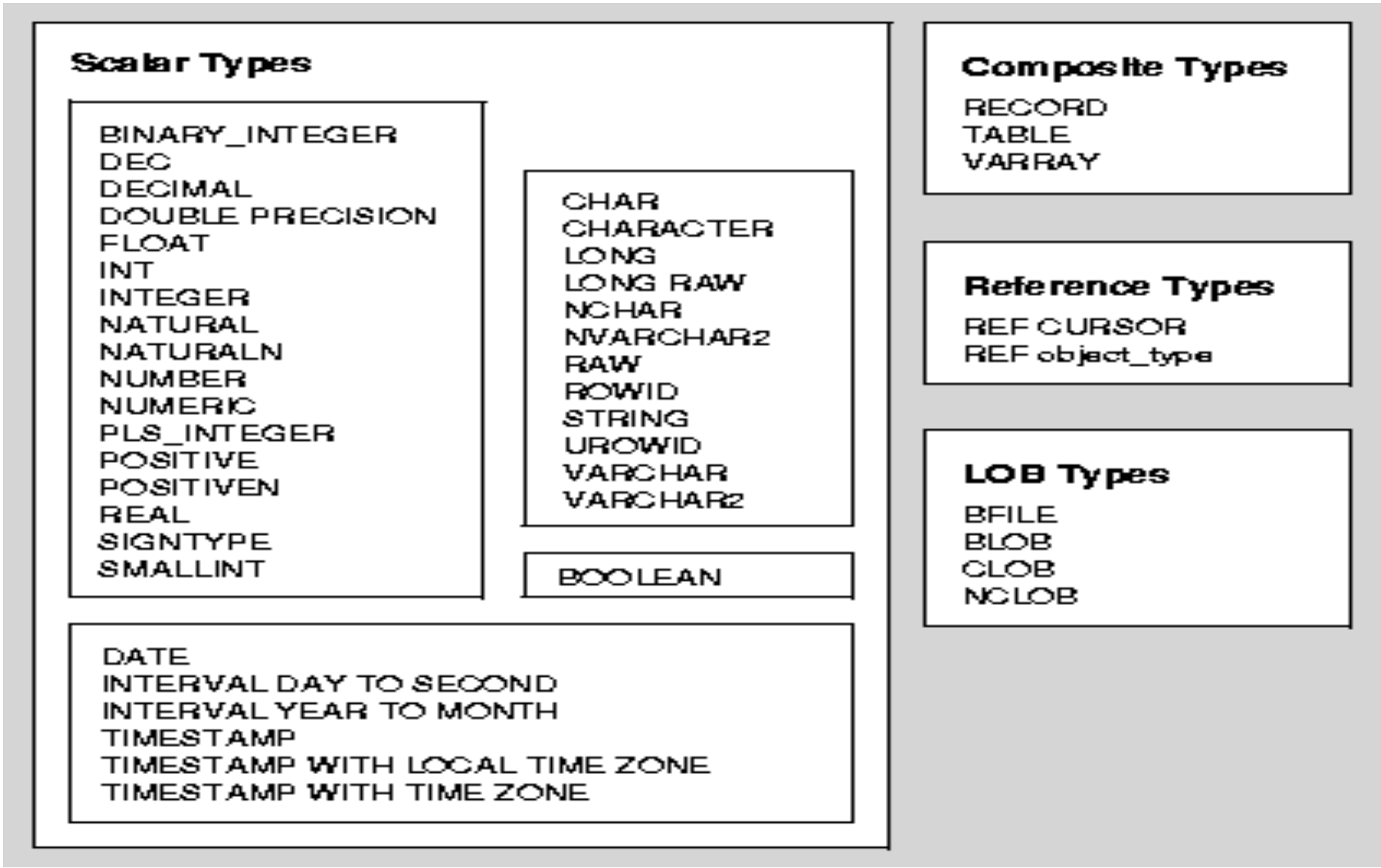
PL/SQL Datatypes

PL/SQL provides a variety of predefined datatypes. In addition, PL/SQL lets you define your own subtypes.

Predefined Datatypes

- A ***scalar*** type has no internal components.
- A ***composite*** type has internal components that can be manipulated individually.
- A ***reference*** type holds values, called pointers, that designate other program items.
- A ***LOB*** type holds values, called lob locators, that specify the location of large objects (graphic images for example) stored out-of-line.

Predefined Datatypes



Composite Datatypes

What Is a Collection?

- A collection is an ordered group of elements, all of the same type
- Each element has a unique subscript that determines its position in the collection.
- PL/SQL offers two kinds of collections:
  - nested tables and
  - varrays (short for variable-size arrays).
- Collections can have only one dimension and must be indexed by integers.

Composite Datatypes

Nested Tables- One-column database tables

- Items of type TABLE are called nested tables
- Oracle stores the rows of a nested table in no particular order. But, when you retrieve the nested table into a PL/SQL variable, the rows are given consecutive subscripts starting at 1.

Varray

- Items of type VARRAY are called varrays.
- They allow you to associate a single identifier with an entire collection.
- This association lets you manipulate the collection as a whole and reference individual elements easily.
- A varray has a maximum size, which you must specify in its type definition.

Composite Datatypes

Updating individual elements in a collection

- Nested Tables: Here this can be achieved using the “THE” operator
- Varray: Here in order to modify an individual element we have to use PL/SQL
  - e.g.: Adding a new element to an existing collection

```
DECLARE
    dept_no         NUMBER;
    new_project     Project;
    position        NUMBER;
    my_projects     ProjectList;
BEGIN
  /* Retrieve project list into local varray. */
  SELECT projects INTO my_projects FROM department
    WHERE dept_no = dept_id FOR UPDATE OF projects;
```

Composite Datatypes

```
 /* Extend varray to make room for new project. */
   my_projects.EXTEND;
   /* Move varray elements forward. */
   FOR i IN REVERSE position..my_projects.LAST - 1 LOOP
      my_projects(i + 1) := my_projects(i);
   END LOOP;
   /* Insert new project. */
   my_projects(position) := new_project;
   /* Update department table. */
   UPDATE department SET projects = my_projects
   WHERE dept_no = dept_id;
END;
```

VARRAYS versus Nested Tables

Varrays have a maximum size, but nested tables do not.

Varrays are always dense, but nested tables can be sparse. So, you can delete individual elements from a nested table but not from a varray.

Oracle stores varray data in-line (in the same tablespace). But, Oracle stores nested table data out-of-line in a store table, which is a system-generated database table associated with the nested table.

When stored in the database, varrays retain their ordering and subscripts, but nested tables do not.

Records

A record is a group of related data items stored in fields, each with its own name and datatype.

- Suppose you have various data about an employee such as name, salary, and hire date. These items are logically related but dissimilar in type. A record containing a field for each item lets you treat the data as a logical unit. Thus, records make it easier to organize and represent information.

Record Vs %ROWTYPE

- The attribute %ROWTYPE lets you declare a record that represents a row in a database table.
- However, you cannot specify the datatypes of fields in the record or declare fields of your own.
- The datatype RECORD lifts those restrictions and lets you define your own records.

Records

```
DECLARE
  TYPE FlightRec IS RECORD (
    flight_no NUMBER(3), gate CHAR(5), departure CHAR(15), arrival CHAR(15),
    passengers PassengerList);  flight_info FlightRec;
  CURSOR c1 IS SELECT * FROM flights;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO flight_info;
    EXIT WHEN c1%NOTFOUND;
    FOR i IN 1..flight_info.passengers.LAST LOOP
      IF flight_info.passengers(i).seat = 'NA' THEN
        DBMS_OUTPUT.PUT_LINE(flight_info.passengers(i).name);
        RAISE seat_not_available;
      END IF;
    END LOOP;
  END LOOP;
  CLOSE c1;
EXCEPTION
  WHEN seat_not_available THEN ……...
END;
```

LOB Types

LOB Types

- The LOB (large object) datatypes BFILE, BLOB, CLOB, and NCLOB let you store blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) up to four gigabytes in size.
- Allow efficient, random, piece-wise access to the data.
- LOB types store values, called locators, that specify the location of large objects stored in an external file, in-line (inside the row) or out-of-line (outside the row).
- Database columns of type BLOB, CLOB, NCLOB, or BFILE store the locators.
- BLOB, CLOB, and NCLOB data is stored in the database, in or outside the row. BFILE data is stored in operating system files outside the database.

LOB Types

- PL/SQL operates on LOBs through the locators.
  - For example, when you retrieve a BLOB column value, only a locator is returned. Locators cannot span transactions or sessions.
  - So, you cannot save a locator in a PL/SQL variable during one transaction or session, then use it in another transaction or session.
  - To manipulate LOBs, you use the supplied package DBMS_LOB.

LOB types Vs. LONG types

- LOBs (except NCLOB) can be attributes of an object type, but LONGs cannot.
- The maximum size of a LOB is four gigabytes, but the maximum size of a LONG is two gigabytes.
- LOBs support random access to data, but LONGs support only sequential access.

NLS Types

Oracle provides National Language Support (NLS), which lets you process single-byte and multi-byte character data and convert between character sets.

It also lets our applications run in different language environments.

NCHAR and NVARCHAR2 store character strings formed from the national character set.

NCHAR datatype

- Stores fixed-length (blank-padded if necessary) NLS character data.

NVARCHAR2 datatype

- Stores variable-length NLS character data.

User-Defined Subtypes

A subtype is a PL/SQL type that is based on an existing type, but is given a new name

A subtype does not introduce a new type; it merely places an optional constraint on its base type.

For e.g., PL/SQL internally predefines the subtypes CHARACTER and INTEGER as follows:

- SUBTYPE CHARACTER IS CHAR;
  - The subtype CHARACTER specifies the same set of values as its base type CHAR, so CHARACTER is an ***unconstrained*** subtype.
- SUBTYPE INTEGER IS NUMBER(38,0); -- allows only whole numbers
  - The subtype INTEGER specifies only a subset of the values of its base type NUMBER, so INTEGER is a ***constrained*** subtype.

Declarations

Variables and Constants can be declared in the declarative part of any PL/SQL block, subprogram, or package.

Declarations

- allocate storage space for a value
- specify its datatype
- name the storage location so that you can reference it.

Declarations can contain

- DEFAULT
- NOT NULL
- %TYPE
- %ROWTYPE

Declarations - Anchoring

You have two choices when you declare a variable:

- Hard-coding the datatype
- Anchoring the datatype to another structure

Whenever possible, use anchored declarations rather than explicit datatype references

- %TYPE for scalar structures
- %ROWTYPE for composite structures

Hard-Coded Declarations

```
ename VARCHAR2(30);
totsales NUMBER (10,2);
```

Anchored Declarations

```
v_ename emp.ename%TYPE;
totsales pkg.sales_amt%TYPE;

emp_rec emp%ROWTYPE;
tot_rec tot_cur%ROWTYPE;
```

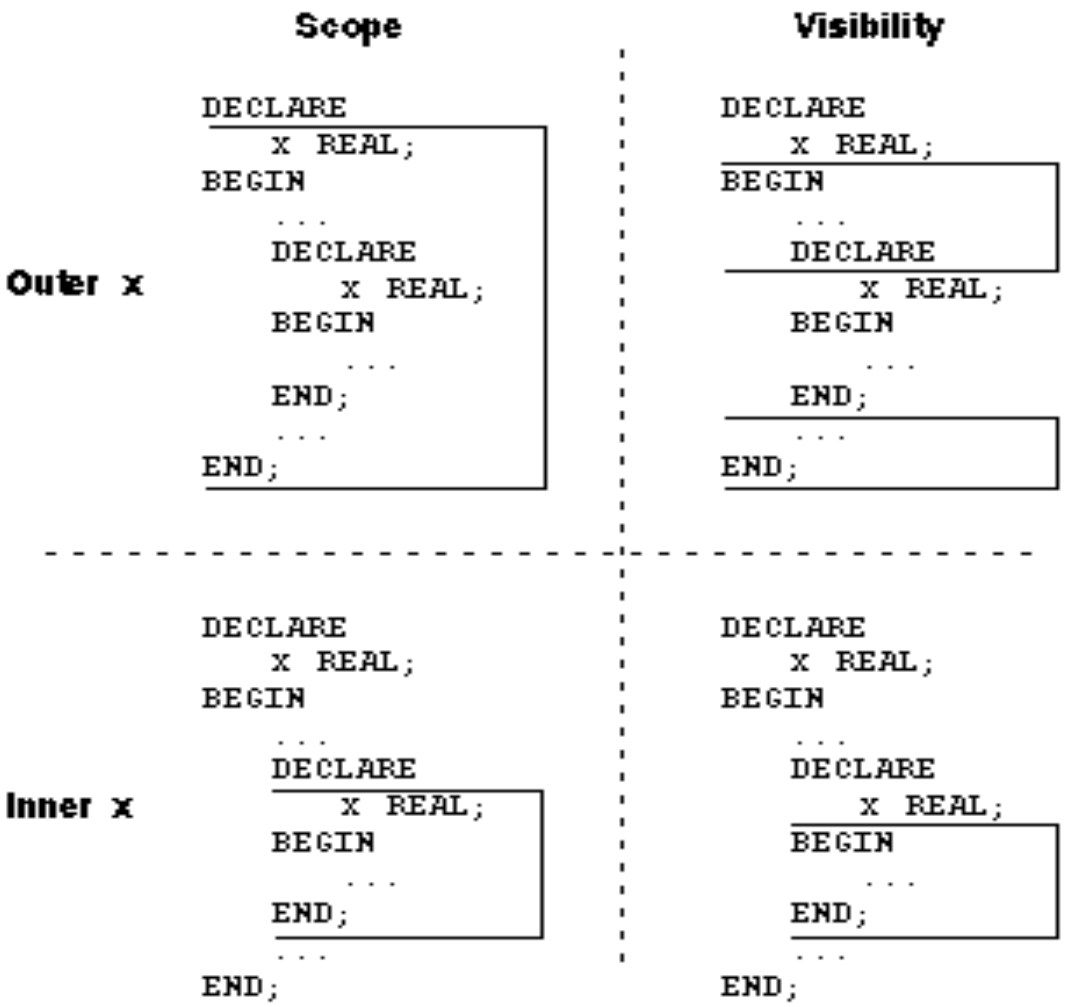
Scope and Visibility

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.

- The following figure shows the scope and visibility of a variable named x, which is declared in an enclosing block, then redeclared in a sub-block.

Scope and Visibility



Control Constructs

Conditional Control

- The selection statement tests a condition and then executes one sequence of statements if the condition is satisfied.

```
IF condition THEN
  sequence_of_statements;
END IF;
```

- EXAMPLE

```
IF sales > quota THEN
    compute_bonus(empid);
    UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

Control Constructs

IF..THEN

```
IF condition THEN
        sequence_of_statements1;
ELSE
        sequence_of_statements2;
END IF;
```

- EXAMPLE

```
IF trans_type = 'CR' THEN
  UPDATE accounts SET balance = balance + credit WHERE …
ELSE
  UPDATE accounts SET balance = balance - debit WHERE …
END IF;
```

Control Constructs

IF..THEN..ELSIF

```
IF <Condition> THEN
    <Action>
ELSIF <Condition>
    THEN   <Action>
 ELSE
    THEN   <Action>
END IF;
```

Control Constructs

Iterative Control

- The Loop statement
- The FOR Loop
- The WHILE Loop
- The GOTO Statement

Control Constructs

The Loop Statement

- Used to execute a sequence of statements a number of times.

```
BEGIN
    ......
    LOOP
            ...
            IF credit_rating < 3   THEN
                    ...
                    EXIT WHEN credit_rating = 0;   -- exit loop immediately
            END IF;
    END LOOP;
END;
```

Control Constructs

The FOR Loop

- The number of iterations through a FOR loop is known before the loop is entered.

```
FOR i IN 1..3 LOOP — assign the values 1,2,3 to i
    sequence_of_statements; — executes three times
END LOOP;
```

Reverse loop in FOR

- With the reverse option Iteration proceeds from higher bound to lower bound.

```
FOR i IN REVERSE 1..3 LOOP — assign the values 3,2,1 to i
    sequence_of_statements; — executes three times
END LOOP;
```

Control Constructs

The WHILE Loop

- The number of iterations through a WHILE loop is not known before entering the loop.

```
WHILE i  IS NOT NULL LOOP
    ...
    IF sal _tab(i)  > 5000  THEN
      RAISE over_limit;
    END IF;
END LOOP;
```

Control Constructs

GOTO statements

- The GOTO statement allows you to continue program processing at a specific label in your program.

```
IF  <condition>  THEN
    GOTO  order_loop;
END IF;
.....
<<order_loop>>
LOOP
.....
END LOOP;
```

Cursors

A PL/SQL construct called a cursor lets you name a work area and access its stored information.

Cursors are of two types

- Explicit Cursors
- Implicit Cursors

Cursor Attributes

Cursors have four attributes that can be effectively used to access the cursor's context area.

They are:

- %ROWCOUNT: The number of rows processed by a SQL statement.
- %FOUND: TRUE if at least one row was processed.
- %NOTFOUND: TRUE if no rows were processed.
- %ISOPEN: TRUE if cursor is open or FALSE if cursor has not been opened or has been closed. Only used with explicit cursors.

Explicit Cursors

SELECT statements which return multiple records inside a PL/SQL block can be declared as a cursor.

Cursors are controlled through four command statements.

They are:

- CURSOR IS: Defines the name and structure of the cursor together with the SELECT statement that will populate the cursor with data. The query is validated but not executed.
- OPEN: Executes the query that populates the cursor with rows.
- FETCH: Loads the row addressed by the cursor pointer into variables and moves the cursor pointer on to the next row ready for the next fetch.
- CLOSE: Releases the data within the cursor and closes it. The cursor can be reopened to refresh its data.

Explicit Cursors

Example

```
DECLARE
    CURSOR c_emp IS
        SELECT emp_code, salary FROM employee
        WHERE deptno = 20;
BEGIN
    OPEN c_emp;
    LOOP
        FETCH c_emp INTO str_emp_code, num_salary;
        EXIT WHEN c_emp%NOTFOUND;
        UPDATE….
    END LOOP;
    COMMIT;
    CLOSE c_emp;
END;
```

Explicit Cursors

Cursor FOR Loop

- The cursor FOR loop simplifies the coding required as the need for opening , fetching or closing the cursor is not required.

```
DECLARE
    CURSOR c_emp IS
                SELECT emp_code, salary FROM employee
                WHERE deptno = 20;
BEGIN
    FOR emp_rec IN c_emp    /*  Cursor index  */
    LOOP
        UPDATE employee SET salary=emp_rec.salary+(emp_rec.salary*0.5)
        WHERE emp_code = emp_rec.emp_code;
    END LOOP;
    COMMIT;
END;
```

Parameterized Cursors

Passing Parameters

- You use the OPEN statement to pass parameters to a cursor.
- For example, given the cursor declaration

```
DECLARE
        emp_name emp.ename%TYPE;
        salary   emp.sal%TYPE;
        CURSOR c1 (name VARCHAR2, salary NUMBER) IS SELECT ...
```

Any of the following statements opens the cursor:

```
OPEN c1(emp_name, 3000);
OPEN c1('ATTLEY', 1500);
OPEN c1(emp_name, salary);
```

Implicit Cursors

When the executable part of a PL/SQL block issues a SQL command, PL/SQL creates an implicit cursor which has the identifier SQL.

The user cannot open,fetch from or close the implicit cursor. Oracle opens the cursor implicitly.

However cursor attributes can be used to access its context area.

Implicit Cursors

Example

```
DECLARE
    rows_affected CHAR(4);
BEGIN
    UPDATE employee SET salary=salary*0.5 WHERE job='Programmer';
    rows_affected :=TO_CHAR(SQL%ROWCOUNT)
    IF SQL%ROWCOUNT >0 THEN
        DBMS_OUTPUT.PUT_LINE(rows_affected || 'Employee Records Modified
          Successfully');
    ELSE
        DBMS_OUTPUT.PUT_LINE('There are no Employees working as Programmers');
    END IF;
END;
```

Cursor Attr. Values

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
OPEN	BEFORE	Exception	FALSE	Exception	exception
OPEN	AFTER	NULL	TRUE	NULL	0
First	BEFORE	NULL	TRUE	NULL	0
FETCH	AFTER	TRUE	TRUE	FALSE	1
Middle	BEFORE	TRUE	TRUE	FALSE	1
FETCH	AFTER	TRUE	TRUE	FALSE	Data dependent
Last	BEFORE	TRUE	TRUE	FALSE	Data dependent
FETCH	AFTER	FALSE	TRUE	TRUE	Data dependent
CLOSE	BEFORE	FALSE	TRUE	TRUE	Data dependent
	AFTER	Exception	FALSE	exception	exception

REF Cursors

Defining REF CURSOR Types

- Define a REF CURSOR type
- Define a cursor variable of that type
- Example

```
DECLARE
  TYPE  EmpCurType IS REF CURSOR RETURN
  emp%ROWTYPE;
  emp_cur EmpCurType
```

Error Handling

Exceptions

- Identifiers in PL/SQL that are raised during the execution of a block to terminate its action.
- A block is always terminated when PL/SQL raises an exception

We can define your own error handler to capture exceptions and perform some final actions before quitting the block.

If PL/SQL handles the exception within the block then the exception will not propagate out to an enclosing block or environment.

Error Handling

There are two classes of exceptions,

- Predefined - Oracle predefined errors which are associated with specific error codes.
- User-defined - Declared by the user and raised when specifically requested within a block. You may associate a user-defined exception with an error code if you wish.

Predefined Exceptions

Internally defined exceptions:

- NO_DATA_FOUND
- TOO_MANY_ROWS
- VALUE_ERROR
- ROWTYPE_MISMATCH
- DUP_VAL_ON_INDEX
- LOGIN_DENIED
- NOT_LOGGED_ON
- CURSOR_ALREADY_OPEN
- INVALID_CURSOR
- ZERO_DIVIDE
- STORAGE_ERROR
- TIMEOUT_ON_RESOURCE
- INVALID_NUMBER
- PROGRAM_ERROR

Predefined Exceptions

```
DECLARE
        acct_type INTEGER := 7;
BEGIN
        SELECT   price / earnings  INTO pe_ratio
        FROM stocks
        WHERE symbol = 'XYZ';    --might cause division–by–zero error
EXCEPTION
        WHEN ZERO_DIVIDE THEN
                INSERT INTO stats (symbol, ratio) VALUES
                        ('XYZ', NULL);
                COMMIT;
        WHEN OTHERS THEN — handles all other errors
                ROLLBACK;
END; –– exception handlers and block end here
```

User Defined Exceptions

These exceptions are explicitly defined by the user and are called using RAISE statements.

- Example

```
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;
```

Functions in Exceptions

In an exception handler, you can use the built-in functions *SQLCODE* and *SQLERRM* to find out which error occurred and to get the associated error message

```
DECLARE
  err_num NUMBER;
  err_msg VARCHAR2(100);
BEGIN
  ...
EXCEPTION
  WHEN OTHERS THEN
    err_num := SQLCODE;
    err_msg := SUBSTR(SQLERRM, 1, 100);
    INSERT INTO errors VALUES (err_num, err_msg);
END;
```

Advantages of PL/SQL Exceptions

Can handle errors conveniently without the need to code multiple checks.

Improves readability by isolating error handling routines.

Improves reliability.

Subprograms

Subprograms are named PL/SQL blocks that can take parameters and be invoked.

PL/SQL has two types of subprograms

- Procedures
- Functions.

Generally, we use a Procedure to perform an action and a Function to compute a value.

Subprogram Parameter Modes

You use parameter modes to define the behavior of formal parameters.

The three parameter modes are:

- IN Mode
- OUT Mode
- IN OUT Mode

IN Mode

- Default mode
- Passes value to a program
- Formal parameters cannot be assigned a value
- Actual parameter can be a constant,initialized variable,literal or expression

Subprogram Parameter Modes

OUT Mode

- Must be specified
- Returns values to the caller
- Formal parameter acts like an uninitilaized variable
- Formal parameter cannot be used in an expression and must be assigned a value
- Actual parameter must be a variable

IN OUT Mode

- Must be specified
- Passes initial value to subprogram and returns updated value to caller
- Formal parameter acts like an initialized variable.
- Formal parameter should be assigned a value
- Actual parameter must be a variable.

Declaring PL/SQL Subprograms

Subprograms can be declared in any PL/SQL block, subprogram, or package.

A subprogram must be declared before calling it.

Syntax for procedure in PL/SQL block

```
PROCEDURE procedure_name[(parameter[, parameter]...)]
{IS | AS}
  [local declarations]
BEGIN
  executable statements
[EXCEPTION
  exception handlers]
END [name];
```

Overloading Subprograms

PL/SQL lets you overload subprogram names and type methods

We can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

Consider we want to initialize the first n rows in two index-by tables that were declared as follows:

```
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;
BEGIN
  ...
END;
```

Overloading Subprograms

- We can write the following Procedure to initialize the index-by table named hiredate_tab:

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

- And, we can write the next Procedure to initialize the index-by table named sal_tab:

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

Stored Procedures

Stored Procedures are database objects.

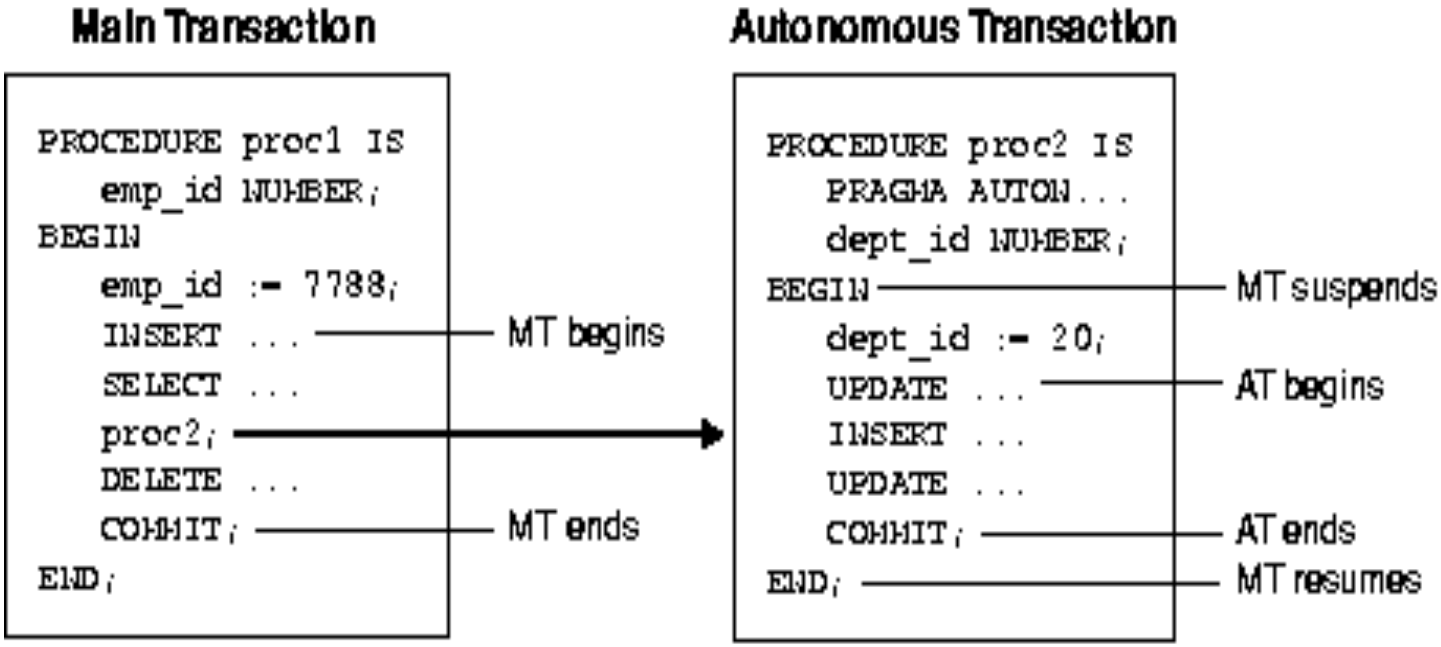
[CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
  [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
  [PRAGMA AUTONOMOUS_TRANSACTION;]
  [local declarations]
BEGIN
  executable statements
[EXCEPTION
  exception handlers]
END [name];
where parameter stands for the following syntax:
parameter_name [IN | OUT [NOCOPY] | IN OUT [NOCOPY]] datatype [{:= |
  DEFAULT} expression]

Stored Procedures

Autonomous Transactions

The pragma AUTONOMOUS_TRANSACTION instructs the PL/SQL compiler to mark a procedure as autonomous (independent).

Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction.



Stored Procedures

NOCOPY Hint

- By default, the OUT and IN OUT parameters are passed by value.
- When the parameters hold large data structures such as collections, records, and instances of object types, all this copying slows down execution and uses up memory.
- To prevent that, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference.

Stored Procedures

```
CREATE 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;
        ELSE
                        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;
```

Note: Composite types such as VARRAYS can also be passed as parameter to Stored Procedure

Stored Functions

A Stored Function is mainly used to compute a value

Stored Functions and Stored Procedures are structured alike, except that Stored Functions have a RETURN clause.

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

Stored Functions

- Example 1:

```
CREATE FUNCTION sal_ok (salary REAL, title VARCHAR2)
   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;
```

- Example 2:

```
CREATE FUNCTION compound (years NUMBER, amount
   NUMBER, rate NUMBER)    RETURN NUMBER IS
BEGIN
            RETURN amount * POWER((rate / 100) + 1, years);
END compound;
```

Advantages of Stored Procedures and Functions

Higher Productivity due to elimination of redundant coding.

Memory Saving. Only one copy of the stored program needs to be loaded into the memory for execution by multiple users.

Application Integrity can be achieved by developing all the applications around a library of stored programs.Coding errors can be reduced.

Tighter Security can be achieved by restricting users to specific database operations by granting access only through subprograms.

Packages

A package is a Database object that groups logically related PL/SQL types , objects and subprograms.

The two parts of a package are:

- PACKAGE SPECIFICATION
- PACKAGE BODY

The specification is the interface to your applications; 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.

Packages

The Package Specification

```
CREATE PACKAGE  emp_actions   AS
        PROCEDURE hire_employee (emp_id INTGER, name VARCHAR2 );
        PROCEDURE fire_employee (emp_id INTEGER);
        PROCEDURE raise_salary (emp_id INTEGER, increase REAL);
...
END emp_actions;
```

Packages

The Package Body

```
CREATE PACKAGE BODY emp_actions AS
    PROCEDURE hire_employee (emp_id INTGER, name VARCHAR2, ...) IS
    BEGIN
        INSERT INTO emp VALUES (empno, ename, ...);
    END hire_employee;
    PROCEDURE fire_employee (emp_id INTEGER) IS
    BEGIN
        DELETE FROM emp WHERE empno = emp_id;
    END fire_employee;
    PROCEDURE raise_salary (emp_id INTEGER, increase REAL) IS salary
      REAL;
    BEGIN
        SELECT sal INTO salary FROM emp WHERE empno = emp_id;
    END raise_salary;
END emp_actions;  /* End Package name */
```

Packages

Advantages of Package

- Modularity due to the encapsulation of logically related types , objects and subprograms in the module.
- Easier Application design : specification can be compiled separately, without its body.
- Information Hiding: Types, objects and subprograms which are to be made public or private can be specified.
- Added functionality: Packaged cursors and variables persist for the duration of the session , so they can be shared by all the Stored Procedures that can execute in the environment.
- Better Performance. When a package subprogram is called for the first time, the whole package gets loaded into the memory. Disk I/O is therefore reduced.

Object Types

An object type encapsulates a data structure along with the functions and procedures needed to manipulate the data.

At run time, when the data structure is filled with values, you have created an object.

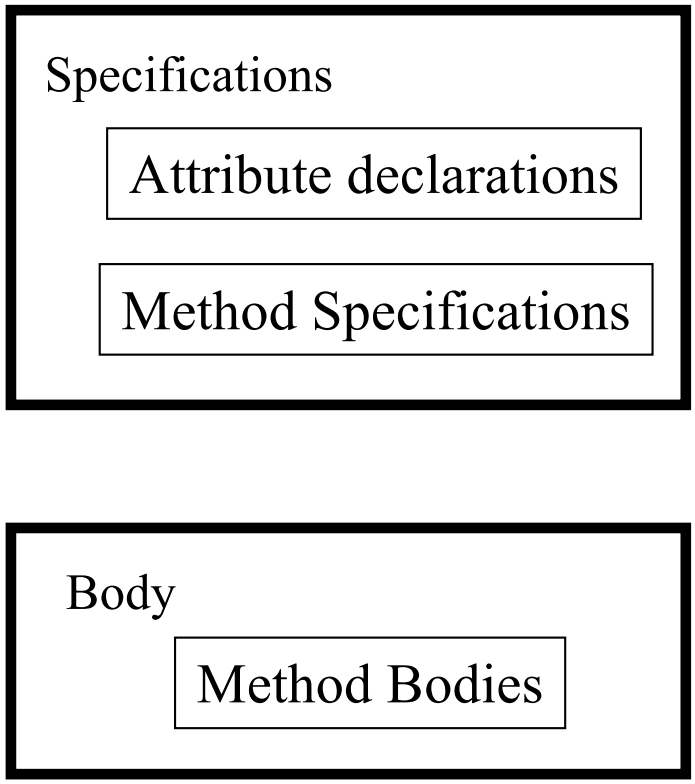
You can create as many objects as you need and each object stores different real-world values.

- The variables that form the data structure are called attributes.
- The Functions and Procedures that characterize the behavior of the object type are called methods.

Structure of an Object Type

- Like a package, an object type has two parts: a specification and a body
- The specification is the interface to your applications; it declares a data structure (set of attributes) along with the operations (methods) needed to manipulate the data.
- The body fully defines the methods, and so implements the specification.

Structure of an Object Type



Public Interface

Private Implementation

Object Type

```
CREATE TYPE Stack AS OBJECT (
    top INTEGER,
    MEMBER FUNCTION full RETURN BOOLEAN,
    MEMBER PROCEDURE push (n IN INTEGER), ... );
CREATE TYPE BODY Stack AS
     ...
    MEMBER PROCEDURE push (n IN INTEGER)
    IS
    BEGIN
    IF NOT full THEN top := top + 1; ...
    END push;
    END;
```

External Procedures

An external procedure is a third-generation-language routine stored in a dynamic link library (DLL), registered with PL/SQL,

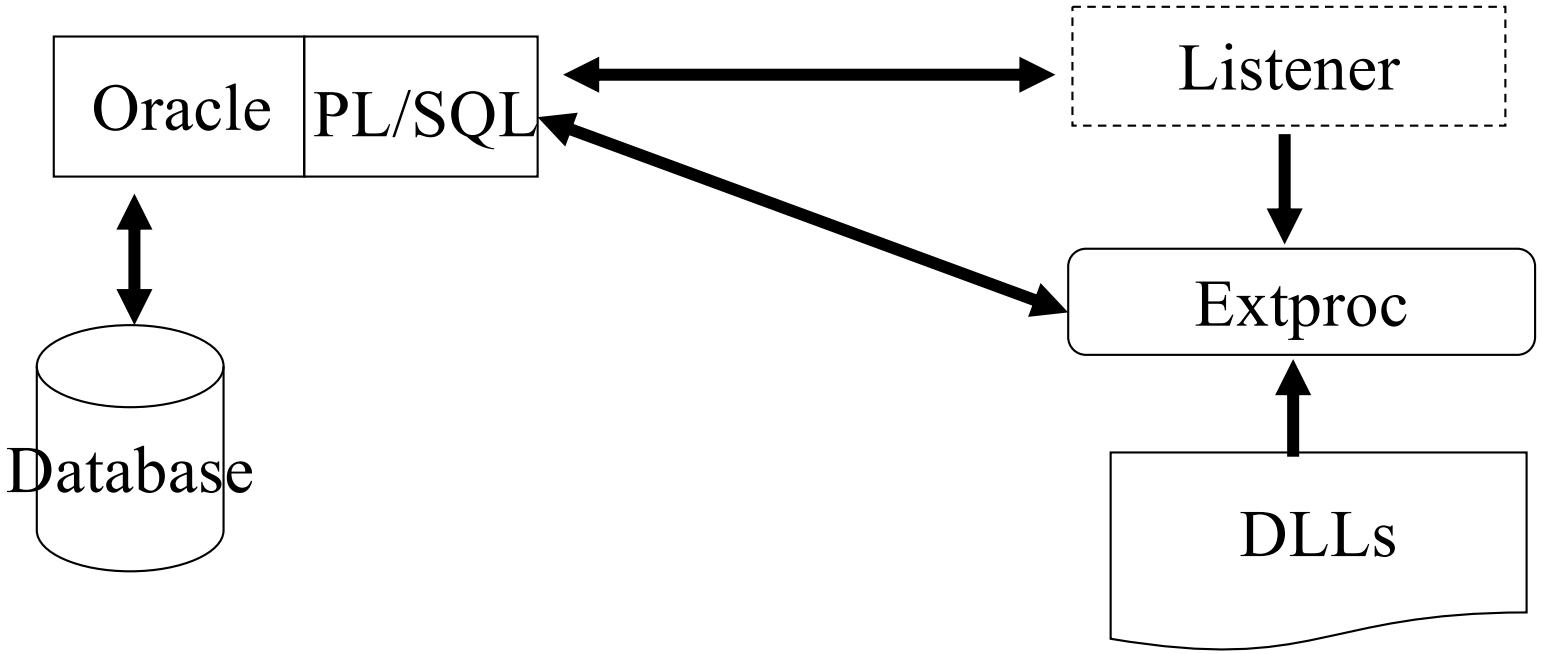
It is called by to do special-purpose processing. The routine must be callable from C but can be written in any language.

How PL/SQL Calls an External Procedure

- To call an external procedure, PL/SQL must know in which DLL it resides. So, PL/SQL looks up the alias library in the EXTERNAL clause of the subprogram that registered the external procedure, then has Oracle look up the DLL in the data dictionary.
- Next, PL/SQL alerts a Listener process, which in turn spawns (launches) a session-specific agent named extproc.

External Procedures

- Then, the Listener hands over the connection to extproc.
- PL/SQL passes to extproc the name of the DLL, the name of the external procedure, and any parameters.
- Then, extproc loads the DLL and runs the external procedure.
- Finally, extproc passes to PL/SQL any values returned by the external procedure. Figure 10-2 shows the flow of control.



External Procedures

Set Up the Environment

DBA sets up the environment for calling external procedures by adding entries to the files tnsnames.ora and listener.ora and by starting a Listener process exclusively for external procedures

Identify the DLL

a DLL is any dynamically loadable operating-system file that stores external procedures

If the DBA grants you CREATE ANY LIBRARY privileges, you can create your own alias libraries using the following syntax:

- **CREATE LIBRARY library_name {IS | AS} 'file_path';**

Create alias library c_utils, which represents DLL utils.so:

- **`create library c_utils as '/DLLs/utils.so';`**

External Procedures

Designate the External Procedure

Find or write a new routine, then add it to the DLL, or simply designate a routine already in the DLL.

Assume that C routine c_gcd, which finds the greatest common divisor of two numbers, is stored in DLL utils.so and that you have EXECUTE privileges on alias library c_utils.

The C prototype for c_gcd follows:

- **int c_gcd(int x_val, int y_val);**

Write a PL/SQL stand-alone function named gcd that registers C routine c_gcd as an external function:

- ```
  CREATE FUNCTION gcd (
          x BINARY_INTEGER,  y BINARY_INTEGER)
  RETURN BINARY_INTEGER AS EXTERNAL
   LIBRARY c_utils
   NAME "c_gcd"  -- quotes preserve lower case
   LANGUAGE C;
  ```
  ```

External Procedures

Register the External Procedure

- **EXTERNAL clause** records information about the external procedure such as its location, its name, the programming language in which it was written, and the calling standard under which it was compiled.
- **EXTERNAL LIBRARY library_name**

  [NAME external_procedure_name]

  [LANGUAGE language_name]

  [CALLING STANDARD {C | PASCAL}]

  [WITH CONTEXT]

  [PARAMETERS (external_parameter[, external_prameter]...)];
- **where external_parameter stands for**

  { CONTEXT

  | {parameter_name | RETURN} [property] [BY REF]
  [external_datatype]}
- **and property stands for**

  {INDICATOR | LENGTH | MAXLEN | CHARSETID | CHARSETFORM}

External Procedures

Calling an External Procedure

- In the example below, you call PL/SQL function gcd from an anonymous block.
- PL/SQL passes the two integer parameters to external function c_gcd, which returns their greatest common divisor.

```
DECLARE
  g BINARY_INTEGER;
  a BINARY_INTEGER;
  b BINARY_INTEGER;
  ...
BEGIN
  ...
  g := gcd(a, b);  -- call function
  IF g IN (2,4,8) THEN ...
```

Triggers

Database Triggers

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

Applications where database triggers are useful

- Verify data integrity on insertion or update
- Implement delete cascade
- Log events transparently
- Enforce complex business rules
- Initiate business process
- Derive column values automatically
- Enforce complex security rules
- Maintain replicated data

Triggers

There are several types of database triggers:

- Triggers are broadly classified as under
  - Statement Level
  - Row Level
- The triggers are listed below

	Row level	Statement level
• Before insert	Y	Y
• After insert	Y	Y
• Before update	Y	Y
• After update	Y	Y
• Before delete	Y	Y
• After delete	Y	Y

Triggers

```
CREATE [OR REPLACE] TRIGGER [schema.]trigger
   {BEFORE event | AFTER event | INSTEAD OF event}
   referencing_clause WHEN (condition) pl_sql_block
```

event can be one or more of the following (separate multiple events with OR)

```
   DELETE event_ref,   INSERT event_ref,   UPDATE event_ref
   UPDATE OF column, column... event_ref
   ddl_statement ON [schema.] {table|view}
   ddl_statement ON DATABASE
   SERVERERROR,   LOGON,   LOGOFF,    STARTUP,   SHUTDOWN
event_ref:
   ON [schema.]table
   ON [schema.]view
   ON [NESTED TABLE nested_table_column OF] [schema.]view
referencing_clause:
   FOR EACH ROW
   REFERENCING OLD [AS] old [FOR EACH ROW]
   REFERENCING NEW [AS] new [FOR EACH ROW]
   REFERENCING PARENT [AS] parent [FOR EACH ROW]
```

Instead Of Triggers

Use INSTEAD OF triggers to perform DELETE, UPDATE, or INSERT operations on views, which are not inherently modifiable

The following view involves a join of two tables and the ability to update records in the view is limited

```
CREATE VIEW worker_lodging_manager
AS
SELECT      worker.name,
            lodging.lodging,
            lodging.manager
FROM        worker,lodging
WHERE       worker.lodging = lodging.lodging
```

Instead Of Triggers

If we use an INSTEAD OF trigger, we can tell Oracle how to update, delete, or insert records in tables

```
CREATE TRIGGER worker_lodging_manager_update
INSTEAD OF UPDATE ON worker_lodging_manager
FOR EACH ROW
BEGIN
        IF :old.name <> :new.name THEN
        UPDATE  worker SET    name = :new.name WHERE   name = :old.name;
        END IF;
        IF :old.lodging <> :new.lodging THEN
        UPDATE  worker SET    lodging = :new.lodging WHERE   name    = :old.name;
        END IF;
        IF :old.lodging <> :new.lodging THEN
        UPDATE  lodging  SET     manager = :new.manager WHERE  lodging  = :old.lodging;
        END IF;
END
```

New Database Triggers - In Oracle 8i

Prior to Oracle 8i, database triggers could be applied to tables only. Essentially, they were table triggers.

Oracle 8i introduces eight new database triggers, which extend beyond previous limitation.

Trigger Event	Executes Before/After	Trigger Description
STARTUP	AFTER	Executes when the database is started
SHUTDOWN	BEFORE	Executes when the database is shut down
SERVERERROR	AFTER	Executes when a server-side error occurs
LOGON	AFTER	Executes when a session connects to the database
LOGOFF	BEFORE	Executes when a session disconnects from the database
CREATE	AFTER	Executes when a database object is created; could be created to apply to the schema or to the entire database
ALTER	AFTER	Executes when a database object is altered; could be created to apply to the schema or to the entire database
DROP	AFTER	Executes when a database object is dropped; could be created to apply to the schema or to the entire database

New PL/SQL Features in Oracle 8i

Native Dynamic SQL

- Oracle 8i introduces the EXECUTE IMMEDIATE command, which provides a much simpler way of creating and executing DDL statements, dynamic SQL, and dynamic PL/SQL as compared to the DBMS_SQL package
- The EXECUTE IMMEDIATE command accepts any SQL statement except SELECT ones that retrieve multiple rows.
- Example

```
CREATE OR REPLACE PROCEDURE Create_Customer
        (Table_Name VARCHAR2, Customer_ID INTEGER,
        Customer_Lastname VARCHAR2,Customer_Firstname VARCHAR2,
        Customer_Address VARCHAR2, Customer_City VARCHAR2,
        Customer_State VARCHAR2,Customer_Zip VARCHAR2,
        Customer_Phone VARCHAR2) IS
cSQL_Statement VARCHAR2(200);
```

New PL/SQL Features in Oracle 8i

```
BEGIN
  cSQL_Statement := 'INSERT INTO ' || LTRIM(RTRIM(Table_Name)) ||
    ' VALUES(:Id, :Last, :First, :Address, :City,
    :State, :Zip, :Phone)';
  EXECUTE IMMEDIATE cSQL_Statement
    USING Customer_ID, Customer_Lastname, Customer_Firstname,
    Customer_Address, Customer_City, Customer_State, Customer_Zip,
    Customer_Phone;
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20101,
      'Error in procedure Create_Customer.');
END Create_Customer;
```

- According to Oracle, Native Dynamic SQL provides 30 to 60 percent performance improvements over DBMS_SQL.

New PL/SQL Features in Oracle 8i

Bulk Binds

- Oracle 8i introduces new PL/SQL FORALL and BULK COLLECT statements to support bulk binds.
  - The FORALL statement is specifically used for processing DML (INSERT, DELETE, and UPDATE) statements to improve performance by reducing the overhead of SQL processing.
  - Example

```
FORALL nCount IN 1..10000
  INSERT INTO Invoices (Invoice_Id, Invoice_Date, Invoice_Amount)
    VALUES (Invoice_Id_Tab(nCount),
            Invoice_Date_Tab(nCount),
            Invoice_Amount_Tab(nCount));
```

  - The equivalent statement for a bulk fetch is the BULK COLLECT clause, which can be used as a part of SELECT INTO, FETCH INTO, or RETURNING INTO clauses:

New PL/SQL Features in Oracle 8i

- Example

```
SELECT Invoice_Id, Invoice_Date, Invoice_Amount
  BULK COLLECT INTO Invoice_Id_Tab, Invoice_Date_Tab, Invoice_Amount_Tab
  FROM Invoice;
```

- The BULK COLLECT clause can be used for both explicit (FETCH INTO) and implicit (SELECT INTO) cursors.
- It fetches the data into the collection (PL/SQL table, varray) starting with element 1 and overwrites all consequent elements until it retrieves all the rows.

- The bulk binds features allow users to increase the performance and reduce the overhead of SQL processing by operating on multiple rows in a single DML statement.
  - The entire collection-not just one collection element at a time-is passed back and forth between the PL/SQL and SQL engines.

New PL/SQL Features in Oracle 8i

Profiler

- An Oracle 8i PL/SQL programmer develops a large number of packages, so the need to identify and solve performance problems becomes critical.
- Oracle 8i provides a profiler that analyzes PL/SQL code and locates bottlenecks.
- The DBMS_PROFILER package is an API that provides the capability to gather statistics related to the execution of the PL/SQL program and identify performance problems.
- The DBMS_PROFILER package is not created by default with the database; we have to generate it with Oracle's ProfLoad.sql script.
- This script has to be executed by the SYS user and access has to be granted to PUBLIC.