PL/SQL

DI Markus Haslinger MSc

DBI/INSY 3rd/4th year

HTL LEONDING

Agenda

1 Introduction
- Introduction

2 Programming Basics
- Basics
- Advanced Variables & Flow Control

3 Advanced Features
- Cursor & Records
- Collections & Exceptions

4 Advanced Programming
- Procedures, Functions & Packages
- Trigger

Motivation

Definition

An application that uses Oracle Database is worthless unless only correct and complete data is persisted. The time-honored way to ensure this is to expose the database only via an interface that hides the implementation details – the tables and the SQL statements that operate on these. This approach is generally called the thick database paradigm, because PL/SQL subprograms inside the database issue the SQL statements from code that implements the surrounding business logic; and because the data can be changed and viewed only through a PL/SQL interface.[1]

[1] https://www.oracle.com/technetwork/database/features/plsql/index.html, 2019-02-06

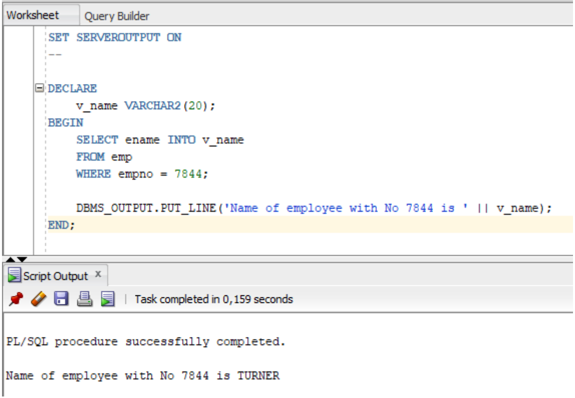
Motivation Discussion

- Access to data should be restricted ⇒ true
- Only correct and complete data should be persisted ⇒ true
- User input has to be validated against business rules ⇒ true
- This business logic should live in the database ⇒ maybe
    - In general these functions are to be performed by the business application...
    - ...because we are using SQL in our application to abstract the type of database used in the backend away
    - **But in certain scenarios PL/SQL can lead to much better performance**

What is PL/SQL?

- PL/SQL is a procedural (programming) language
  - Variables
  - Control structures (loops, branches)
  - Reusability (functions)
  - Exception Handling
- It focuses on intertwining SQL statements and program code
- Execution (and compilation) happens on the DB(MS)
- Can be used everywhere where Oracle is installed

PL/SQL Sample



```
SET SERVEROUTPUT ON
--

DECLARE
    v_name VARCHAR2(20);
BEGIN
    SELECT ename INTO v_name
    FROM emp
    WHERE empno = 7844;

    DBMS_OUTPUT.PUT_LINE('Name of employee with No 7844 is ' || v_name);
END;
```

Task completed in 0,159 seconds

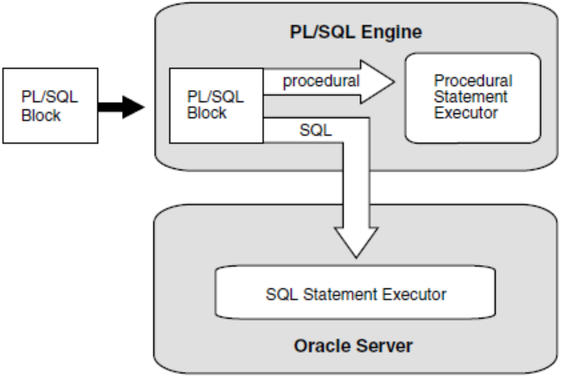
```
PL/SQL procedure successfully completed.

Name of employee with No 7844 is TURNER
```

Interlude: PL/SQL Output in SQLDeveloper

- Use 'SET SERVEROUTPUT ON' or
- Activate DBMS Output View

PL/SQL Execution



Benefits of PL/SQL (vs. pure SQL)

- SQL just defines *what* has to be done, not *how*
- Performance can be improved by reusing results
- Integration in (Oracle) tools (reports, forms,. . .)
- Exception Handling

Block Structure

1. DECLARE (optional)
   - Variables, Cursors, user defined Exceptions
2. BEGIN
   - SQL statements
   - PL/SQL statements
3. EXCEPTION (optional)
   - Exception handling
4. END;

Block Types

Procedure

```
PROCEDURE name
IS
BEGIN
-- statements
[EXCEPTION]
END;
```

Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
-- statements
RETURN value;
[EXCEPTION]
END;
```

Anonymous

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

Variables

- Syntax: identifier [CONSTANT] datatype [**NOT NULL**] [:= | **DEFAULT** expr];
- Identifier
  - Have to start with a letter
  - Can contain letters and numbers
  - Can contain special characters ($,. . .)
  - Have a max. length of 30
  - Must not be reserved words

Variable Examples

Variables

```
v_name VARCHAR2(20);
v_name VARCHAR2(20) := ’Max’;
v_name VARCHAR2(20) DEFAULT ’Susi’;
c_pi CONSTANT NUMBER(3,2) := 3.14;
```

Datatypes

- CHAR [(length)]
- VARCHAR2 (max_length)
- NUMBER [(precision, scale)]
- BINARY_INTEGER
- PLS_INTEGER
- BOOLEAN
- BINARY_FLOAT
- BINARY_DOUBLE
- DATE
- TIMESTAMP

%TYPE datatype declaration

- Allows to automatically assume datatype of a column or another variable
- Datatype changes with the other ⇒ this only works as long as the new datatype is still valid for the performed operations (technically & logically)
- Syntax Examples:
  - identifier **table**.column_name%TYPE;
  - v_balance **NUMBER**(7,2);
    v_min_balance v_balance%TYPE := 500;

Strings

- Single apostrophe
  - Example: 'Hello World'
- Escaping:
  1. double quotes: 'Sorry Dave, I can''t do that'
  2. q-notation: q'<Delimiter>...<Delimiter>'
     - q'!Sorry Dave, I can't do that!'
     - q'[Sorry Dave, I can't do that]'

SQL Functions

- These functions can be used in PL/SQL (outside of a SQL statement)
- Datatype conversions
  - TO_CHAR
  - TO_DATE
  - TO_NUMBER
  - TO_TIMESTAMP
- Sequences
  - var := my_seq.NEXTVAL;
- String functions
  - LENGTH,. . .
- Numerical functions
  - MONTHS_BETWEEN,. . .
- **No** aggregation functions (AVG, MIN, . . .)

Hello World Example

Example

```
SET SERVEROUTPUT ON

DECLARE
    name VARCHAR2(10) := '<your_name>';
    birth DATE := date '<your_birthday>';
    age NUMBER;
BEGIN
    age := FLOOR(MONTHS_BETWEEN(SYSDATE, birth)/12);
    dbms_output.put_line('Hello, my name is ' || name || ' and I am ' || age || '
         years old');
END;
```

SELECT in PL/SQL

- INTO clause
- End each query with a semicolon
- Queries must only return a single result (otherwise a CURSOR is needed)

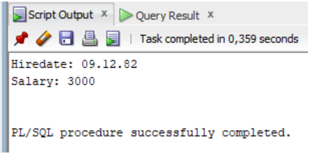
Syntax

```
SELECT <what>
INTO <variable>
FROM <table>
[WHERE <condition>];
```

SELECT in PL/SQL – Example

Example

```
SET SERVEROUTPUT ON
DECLARE
    v_hiredate emp.hiredate%TYPE;
    v_salary emp.sal%TYPE;
BEGIN
    SELECT hiredate, sal
    INTO v_hiredate, v_salary
    FROM emp
    WHERE empno = 7788;
    dbms_output.put_line('Hiredate: ' ||
            v_hiredate);
    dbms_output.put_line('Salary: ' ||
            v_salary);
END;
```



```
Hiredate: 09.12.82
Salary: 3000


PL/SQL procedure successfully completed.
```

DML in PL/SQL – Examples

INSERT

```
BEGIN
    INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
    VALUES (9999, 'DMC', 'Magician', 7839, date '2019-02-22', 1337, null, 40);
END;
```

UPDATE

```
DECLARE
    v_raise emp.sal%TYPE := 200;
BEGIN
    UPDATE emp SET sal = sal + v_raise WHERE empno = 9999;
END;
```

DML in PL/SQL – Examples

DELETE

```
DECLARE
    v_deptno emp.deptno%TYPE := 40;
BEGIN
    DELETE FROM emp WHERE deptno = v_deptno;
END;
```

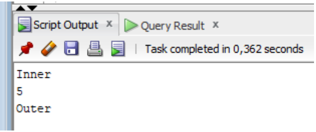
HTL LEONDING

Nested Scopes/Blocks

- Blocks can be nested
- Variable lookup start at the innermost block

Example

```
SET SERVEROUTPUT ON
DECLARE
    v_pseudo_global VARCHAR2(10) := 'Outer';
BEGIN
    DECLARE
        v_pseudo_global NUMBER := 5;
        v_inner VARCHAR2(10) := 'Inner';
    BEGIN
        dbms_output.put_line(v_inner);
        dbms_output.put_line(v_pseudo_global);
    END;
    dbms_output.put_line(v_pseudo_global);
END;
```



Task completed in 0,362 seconds

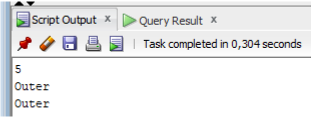
```
Inner
5
Outer
```

Nested Scopes/Blocks – Qualifier

- Access outer variables explicitly

Example

```
SET SERVEROUTPUT ON
BEGIN <<OUTER>>
    DECLARE
        v_pseudo_global VARCHAR2(10) := 'Outer';
    BEGIN
        DECLARE
            v_pseudo_global NUMBER := 5;
        BEGIN
            dbms_output.put_line(v_pseudo_global);
            dbms_output.put_line(OUTER.v_pseudo_global);
        END;
        dbms_output.put_line(v_pseudo_global);
    END;
END OUTER;
```



Task completed in 0,304 seconds

```
5
Outer
Outer
```

Variables – BIND

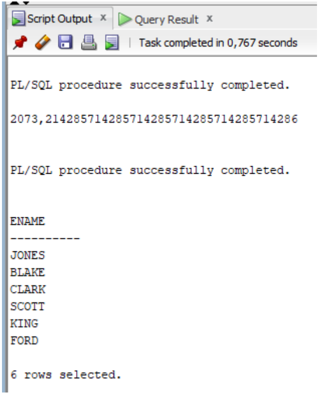
- Syntax: VARIABLE name type
- Also called Hostvariables (global variables)
- Referenced by a leading colon
- Can be used in SQL statements and PL/SQL blocks
- Still availabe after execution of the PL/SQL block
- Output via PRINT
  - **SET** AUTOPRINT **ON**

Variables – BIND – Example

Example

```
SET SERVEROUTPUT ON

VARIABLE b_avgsal NUMBER
BEGIN
    SELECT AVG(sal) INTO :b_avgsal
    FROM emp;
END;
/
BEGIN
    dbms_output.put_line(:b_avgsal);
END;
/
SELECT ename FROM emp
WHERE sal > :b_avgsal;
```



Task completed in 0,767 seconds

```
PL/SQL procedure successfully completed.

2073,214285714285714285714285714286


PL/SQL procedure successfully completed.


ENAME
----------
JONES
BLAKE
CLARK
SCOTT
KING
FORD

6 rows selected.
```

IF Conditional

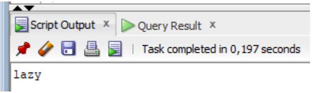
Syntax

```
IF <condition> THEN
    <statements>;
[ELSIF <condition> THEN
    <statements>;]
[ELSE
    <statements>;]
END IF;
```

IF Conditional – Example

Syntax

```
DECLARE
    v_score INTEGER := 55;
BEGIN
    IF v_score < 50 THEN
        dbms_output.put_line('failed in life');
    ELSIF v_score < 60 THEN
        dbms_output.put_line('lazy');
    ELSE
        dbms_output.put_line('promising');
    END IF;
END;
```



Task completed in 0,197 seconds

```
lazy
```

CASE Expression

- Similar to switch, but still different
- Two versions: including & omitting the expression clause

Syntax

```
CASE [expression]
    WHEN condition_1 THEN result_1
    WHEN condition_2 THEN result_2
    [WHEN condition_n THEN result_n]
    [ELSE result_n+1]
END;
```

CASE Expression – Example

Example

```
DECLARE
    v_perc INTEGER := 82;
    v_grade VARCHAR2(20);
BEGIN
    v_grade := CASE
        WHEN v_perc >= 92.0 THEN ’Sehr Gut’
        WHEN v_perc >= 80.0 THEN ’Gut’
        WHEN v_perc >= 65.0 THEN ’Befriedigend’
        WHEN v_perc >= 50.0 THEN ’Genuegend’
        ELSE ’Nicht Genuegend’
    END;
    dbms_output.put_line(v_grade);
END;
```

CASE Expression – Example with SQL

With Expression

```
SELECT table_name,
CASE owner
     WHEN ’SYS’ THEN ’The owner
           is SYS’
     WHEN ’SYSTEM’ THEN ’The
           owner is SYSTEM’
     ELSE ’The owner is another value
           ’
END
FROM all_tables;
```

Without Expression

```
SELECT table_name,
CASE
     WHEN owner=’SYS’ THEN ’The owner
           is SYS’
     WHEN owner=’SYSTEM’ THEN ’The
           owner is SYSTEM’
     ELSE ’The owner is another value’
END
FROM all_tables;
```

CASE Statement

- Does not return value, but executes specific action
- Ends with **END CASE**

Example

```
DECLARE
    v_grade VARCHAR2(20) := ’3’;
BEGIN
    CASE v_grade
        WHEN ’1’ THEN dbms_output.put_line(’Sehr Gut’);
        WHEN ’2’ THEN dbms_output.put_line(’Gut’);
        WHEN ’3’ THEN dbms_output.put_line(’Befriedigend’);
        WHEN ’4’ THEN dbms_output.put_line(’Genuegend’);
        ELSE dbms_output.put_line(’Nicht Genuegend’);
    END CASE;
END;
```

Loops – WHILE

- A typical while loop

Syntax

```
WHILE condition LOOP
    statement_1;
    [statement_n;]
END LOOP;
```

Loops – LOOP

- Similar to a do-while loop
- Runs forever until EXIT (with optional condition) is hit

Syntax

```
LOOP
    statement_1;
    [statement_n;]
    EXIT [WHEN condition];
END LOOP;
```

HTL LEONDING

Loops – FOR

- Pretty typical for loop
- Supports range syntax and reverse option

Syntax

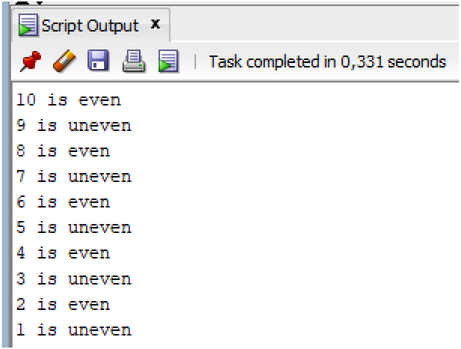
```
FOR counter IN [REVERSE] lower_bound..upper_bound LOOP
    statement_1;
    [statement_n;]
END LOOP;
```

Loops – FOR – Example

Example

```
DECLARE
    v_even BOOLEAN;
BEGIN
    FOR i IN REVERSE 1..10 LOOP
        v_even := i MOD 2 = 0;
        dbms_output.put_line(i || ' is ' || CASE WHEN v_even =
             TRUE THEN 'even' ELSE 'uneven' END);
    END LOOP;
END;
```

LOOPS – FOR – Example Result



```
10 is even
9 is uneven
8 is even
7 is uneven
6 is even
5 is uneven
4 is even
3 is uneven
2 is even
1 is uneven
```

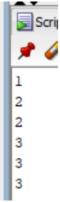
Nested Loops

- Loops can be nested within other Loops
    - Of course the same is true for IFs within IFs and IFs within LOOPs
- Like blocks, they can be labeled

Nested Loops – Example

Example

```
DECLARE
    v_cnt NUMBER := 0;
BEGIN
    <<OUTER_LOOP>>
    LOOP
        v_cnt := v_cnt + 1;
    EXIT WHEN v_cnt > 5;
        <<INNER_LOOP>>
        FOR i IN 1..v_cnt LOOP
            EXIT OUTER_LOOP WHEN v_cnt > 3 AND v_cnt > i;
            dbms_output.put_line(v_cnt);
        END LOOP INNER_LOOP;
    END LOOP OUTER_LOOP;
END;
```



```
1
2
2
3
3
3
```

Loops – CONTINUE

- Skips the remaining statements of the current iteration
- Syntax: CONTINUE [label] [**WHEN** condition]

Examples

```
IF x < 3 THEN
    CONTINUE;
END IF;
```

or

```
CONTINUE WHEN x < 3;
```

Cursor

- A cursor is a pointer to a private memory block assigned by the DBMS
    - Similar to a C Array pointer
- It is used to process result sets (= more than 1 result)
- There are two types:
    - Implicit Cursor: Created automatically when a SQL query is executed
    - Explicit Cursor: Defined manually by the programmer

Implicit Cursor Attributes

- SQL%ROWCOUNT
- SQL%[NOT]FOUND

Example

```
SET SERVEROUTPUT ON
BEGIN
    DELETE FROM dummy WHERE dummy = 3;
    dbms_output.put_line(SQL%ROWCOUNT || ’ rows deleted’);
END;
```

Explicit Cursor

- Declared and managed by the programmer
  - Contrary to the implicit cursor which is created and managed automatically behind the scenes
- Used for SELECT statements which return multiple rows
- Allow for the row-wise processing of result sets
- Always four steps:
  1. Declaring the cursor in the DECLARE section
  2. Opening the cursor executes the query and locks the rows
  3. Reading the data
     - FETCH each row
     - until %NOTFOUND becomes true indicating no more rows
  4. Closing the cursor (CLOSE) releases the rows

Declaring a Cursor

- Syntax: **CURSOR** cursor_name **IS** select_statement
- The command INTO must not be used in the cursor declaration but is used later when FETCHing
- Using variables in the SELECT is possible

Example

```
v_max_player_no NUMBER := 100;
CURSOR c_players IS
    SELECT * FROM players WHERE playerno <=
        v_max_player_no;
```

Explicit Cursor – Example

Iterating players

```
DECLARE
    v_max_player_no NUMBER := 100;
    CURSOR c_players IS
        SELECT * FROM players WHERE playerno <= v_max_player_no;
    v_player players%ROWTYPE;
BEGIN
    OPEN c_players;
    LOOP
        FETCH c_players INTO v_player;
        EXIT WHEN c_players%NOTFOUND;
        dbms_output.put_line(v_player.name);
    END LOOP;
    CLOSE c_players;
END;
```

Cursor – FOR Loops

- A cursor is basically an iterator
- Using it in a 'foreach' loop simplifies the process
  - OPEN, FETCH, CLOSE and the check for more rows are done automatically
  - Iteration variable record declared implicitly

Syntax

```
FOR record_name IN cursor_name LOOP
    ...
END LOOP;
```

Cursor – FOR Loops – Example

Example

```
DECLARE
    v_max_player_no NUMBER := 100;
    CURSOR c_players IS
        SELECT * FROM players WHERE playerno <=
            v_max_player_no;
BEGIN
    FOR v_player IN c_players LOOP
        dbms_output.put_line(v_player.name);
    END LOOP;
END;
```

Cursor Attributes

- Usually used when not iterating via a FOR loop

Attribute	**Type**	**Description**
%ISOPEN	BOOLEAN	TRUE if cursor is opened
%NOTFOUND	BOOLEAN	TRUE if last FETCH did not return a row
%FOUND	BOOLEAN	Opposite of %NOTFOUND
%ROWCOUNT	NUMBER	Count of rows returned so far (*not* total rows in the result set)

Omitting Cursor declaration

- Declaration of the cursor can be omitted

Example

```
DECLARE
    v_max_player_no NUMBER := 100;
BEGIN
    FOR v_player IN (SELECT * FROM players WHERE playerno
         <= v_max_player_no) LOOP
        dbms_output.put_line(v_player.name);
    END LOOP;
END;
```

Cursor – Parameters

- A cursor can be defined with parameters
- These parameters are supplied when opening the cursor
- This allows the same cursor to be used for several similar queries

Syntax

```
CURSOR cursor_name [(parameter_name datatype,...)] IS
    select_statement;
```

Cursor – Parameters – Example

Example

```
DECLARE
    CURSOR c_players (p_max_playno NUMBER) IS
        SELECT * FROM players WHERE playerno <=
            p_max_playno;
BEGIN
    FOR v_player IN c_players(100) LOOP
        dbms_output.put_line(v_player.name);
    END LOOP;
END;
```

Cursor – FOR UPDATE

- A DBMS is meant to be used by multiple users at the same time
- Thus it is important to deal with several, different updates occurring at the same time
- Normally, the data set a cursor iterates is a snapshot
  - So changes made while iterating are not visible
- FOR UPDATE locks the affected rows for updates
- This is important if changes are made based on the row values
  - Otherwise the decision may be based on outdated values

Cursor – FOR UPDATE

Syntax

```
SELECT ... FROM ...
FOR UPDATE [OF column_ref][NOWAIT | WAIT n];
```

- column_ref: one or more columns (to lock)
- NOWAIT: raises an error of the rows are locked by another session already
- WAIT: waits for the specified number of seconds for the lock to release (before raising the error)

Cursor – WHERE CURRENT OF

- Allows to UPDATE or DELETE the current row the cursor points to
- This simplifies the process, because no full primary key condition has to be used
- Usually used together with FOR UPDATE to avoid incorrect changes

Cursor – WHERE CURRENT OF – Example

Example

```
DECLARE
    CURSOR c_players (p_max_playno NUMBER) IS
        SELECT * FROM players WHERE playerno <= p_max_playno FOR
             UPDATE;
BEGIN
    FOR v_player IN c_players(50) LOOP
        IF v_player.town = ’Stratford’ THEN
            UPDATE players SET town=’Bradford’
            WHERE CURRENT OF c_players;
        END IF;
        dbms_output.put_line(v_player.name || ’ ’ || v_player.town);
    END LOOP;
END;
```

Record

- Grouping values of different datatypes together
- Similar to a C struct

Syntax

```
TYPE type_name IS RECORD
(field_declaration [, field_declaration]);

−−Usage
identifier type_name;
```

Record – Example

Example

```
SET SERVEROUTPUT ON;
DECLARE
    TYPE rt_empdata IS RECORD
    (
        emp_no NUMBER(6) NOT NULL := −1,
        salary NUMBER(8,2) NOT NULL := −1,
        comm NUMBER(8,2) NULL
    );
    v_emp rt_empdata;
BEGIN
    v_emp.salary := 2200;
    dbms_output.put_line(v_emp.salary);
END;
```

HTL LEONDING

Record – %ROWTYPE

- Syntax: identifier reference%ROWTYPE
- Declares a record which can hold a whole row of the table
- Can be used for INSERT and UPDATE

Example

```
DECLARE
    v_player players%ROWTYPE;
BEGIN
    SELECT * INTO v_player FROM players WHERE playerno=44;
    dbms_output.put_line(v_player.postcode);
END;
```

Collections

- Can contain multiple values of the same data type
- Comparable to an array
- Three types exist:
  - VARRAY: a pretty standard array
  - Associative array: comparable to a dictionary
  - Nested Table: associative array without INDEX BY definition, extendable and a little like a list

VARRAY

- Syntax: TYPE typename IS VARRAY(**size**)OF datatype;
- Has a fixed size
- Element order is maintained
- Has only one datatype
  - Including the max size (e.g. VARCHAR2(10))
- Array is 1 based, not 0 based as usual!

VARRAY – Example

Example

```
DECLARE
    TYPE at_numbers IS VARRAY(2) OF NUMBER(1);
    v_num_arr at_numbers;
BEGIN
    v_num_arr := at_numbers(2, 4);
    FOR i in 1..v_num_arr.count LOOP
        dbms_output.put_line(v_num_arr(i));
    END LOOP;
END;
```

Associative Array

- Key Value Pairs (like a Dictionary)
- The Key
    - Can be a numeric or string value
    - Instead of NUMBER use BINARY_INTEGER or PLS_INTEGER
    - VARCHAR2 (and DATE)
- The Value
    - Scalar datatype or
    - Record datatype ⇒ nesting is possible

Associative Array

Syntax

```
TYPE typename IS TABLE OF (value_type [NOT NULL] | table%ROWTYPE)
    INDEX BY (PLS_INTEGER | BINARY_INTEGER | VARCHAR2(size))
```

Example

```
DECLARE
    TYPE day_of_week IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;
    days day_of_week;
BEGIN
    days(1) := 'Monday';
    days(2) := 'Tuesday';
    dbms_output.put_line(CASE WHEN days.EXISTS(2) THEN days(2) ELSE 'unknown' END);
END;
```

Associative Array – Functions

- EXISTS(n): Returns true if the nth element exists ⇒ looks for the index (not key value per-se) which can be problematic when mixing numeric and alphanumeric values
- COUNT: Returns the number of elements in the array
- FIRST: Returns the smallest index number or NULL if the array is empty
- LAST: Returns the biggest index number or NULL if the array is empty

Associative Array – Functions

- PRIOR(n): Returns the index number preceding the index n in the array
- NEXT(n): Returns the index number succeeding the index n in the array
- DELETE
  - DELETE: Removes all elements from the array
  - DELETE(n): Removes the n^{th} element from the array
  - DELETE(m,n): Removes all elements between the m^{th} and n^{th}

Associative Array – %ROWTYPE Example

Example

```
DECLARE
    TYPE tt_players IS TABLE OF players%ROWTYPE INDEX BY
         PLS_INTEGER;
    v_players tt_players;
BEGIN
    SELECT * INTO v_players(44) FROM players WHERE playerno
        = 44;
    dbms_output.put_line(v_players(44).postcode);
END;
```

Nested Table

- Similar to Associative Table without an INDEX
- Has to be initialized with a constructor
- Has only positive indices
- Could be saved in the database (valid datatype for schema tables)
- Be careful:
  - There is no fixed upper size as with a VARRAY
  - But you still need to extend every time you want to add a row

Nested Table – Example

Example

```
DECLARE
    TYPE tt_dept IS TABLE OF VARCHAR2(20);
    departments tt_dept;
BEGIN
    departments := tt_dept(’Informatik’, ’Elektronik’, ’Medizintechnik’);
    departments.extend(1);
    departments(4) := departments(3);
    departments(3) := ’Medientechnik’;
    FOR i in 1..departments.COUNT LOOP
        dbms_output.put_line(departments(i));
    END LOOP;
END;
```

Exceptions

- Not compile errors, but runtime exceptions
- Raised either implicitly by the database server or explicitly in the code
- Handling:
  - Either catch and handle or
  - Propagating to the caller ← we rarely want that

Type	Description	To declare	Raised
Predefined Oracle Error	The ∼20 most common errors	No	Implicitly
Non-predefined Oracle Errors	All other default errors	Yes	Implicitly
User defined Error		Yes	Explicitly

Exception

Syntax

```
EXCEPTION
    WHEN exception1 THEN ...
    [WHEN exceptionN THEN ...]
    [WHEN OTHERS THEN ...]
```

- **WHEN** OTHERS can be used to catch all unexpected Exceptions
- For a list of predefined Oracle Exceptions see `https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/07_errs.htm#784`

Exception – Example

Example

```
DECLARE
    v_name VARCHAR2(20);
BEGIN
    SELECT name INTO v_name FROM players;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        dbms_output.put_line('Select with more than 1 result!');
END;
```

Exception – Not Predefined

Declaration

```
DECLARE
    exception EXCEPTION;
    PRAGMA EXCEPTION_INIT (exception, errorno);
```

Handling

```
EXCEPTION
    WHEN exception THEN
        ...
```

Exception – Not Predefined – Example

Example

```
DECLARE
    e_insert_null EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_insert_null, −01400);
BEGIN
    INSERT INTO players (playerno) VALUES (NULL);
EXCEPTION
    WHEN e_insert_null THEN
        dbms_output.put_line(’No NULL in NOT NULL column’);
        dbms_output.put_line(SQLERRM);
END;
```

Exception – Functions

- SQLERRM
  - Returns the message associated with the error code
- SQLCODE
  - Returns the numeric value of the error code
  - Assignable to a NUMBER variable
- If one of the return values is to be used in a SQL statement it has to be assigned to a variable first.

User defined Exceptions

Declaration

```
DECLARE
    exception EXCEPTION;
```

Raising & Handling

```
-- first raise when appropriate
RAISE exception;

-- then handle
EXCEPTION
    WHEN exception THEN ...
```

User defined Exceptions – Example

Example

```
DECLARE
    e_invalid_cust_no EXCEPTION;
    v_cust_no NUMBER := 15000;
BEGIN
    IF v_cust_no > 9999 THEN
        RAISE e_invalid_cust_no;
    END IF;
EXCEPTION
    WHEN e_invalid_cust_no THEN
        dbms_output.put_line(’Invalid customer number’);
END;
```

RAISE_APPLICATION_ERROR

Syntax

RAISE _APPLICATION_ERROR
    (error_number, message[, (**TRUE** | **FALSE**)]);

- Returns user defined errors to the (calling) application
- error_number: Number between -20999 and -20000
- message: User defined error message (max. 2kB)
- TRUE vs. FALSE
    - TRUE: Keeps previous error in stack
    - FALSE: Replaces previous error in stack

Subroutines & Functions

- We differentiate between:
  - Procedures – no return value
  - Functions – return value
- Are basically named PL/SQL blocks
  - Which can accept *parameters*
  - Structure similar, with:
    - Declaration section (optional, *no* DECLARE)
    - Statement section (required)
    - Exception Handling (optional)
- Are compiled once and then stored in the database
  - ⇒ 'Stored procedure'
  - Anonymous blocks are compiled at every execution

Procedures

Syntax

```
CREATE [OR REPLACE] PROCEDURE proc_name
[(
      argument1 [mode1] datatype1,
      argument2 [mode2] datatype2,
      ...
)] (IS|AS)
proc_body;
```

Procedures – Parameters

- Three different modes:
  - IN (default): regular input parameter
  - OUT: out parameter
  - IN OUT: both input and output
- Data types:
  - The data type of a parameter must not have a specific size defined (CHAR vs. CHAR(5))
  - If tailoring to a specific table %TYPE can be used

HTL LEONDING

Procedures – Example

Example

```
CREATE OR REPLACE PROCEDURE add_employee (
    p_ssn employee.ssn%TYPE,
    p_name employee.name%TYPE,
    p_city employee.city%TYPE)
AS
BEGIN
    INSERT INTO employee(ssn, name, exitdate, city)
    VALUES (p_ssn, p_name, NULL, p_city);
    dbms_output.put_line(SQL%ROWCOUNT || ' row inserted');
END;
```

Procedures

How to call

```
BEGIN
    add_employee('9876080888', 'new emp', 'Sto Lat');
END;
```

Retrieve object info

```
SELECT * from user_objects WHERE object_name = '
    ADD_EMPLOYEE';
SELECT * from user_source WHERE name = 'ADD_EMPLOYEE';
```

Functions

Syntax

```
CREATE [OR REPLACE] FUNCTION func_name
[(
    argument1 [mode1] datatype1,
    argument2 [mode2] datatype2,
    ...
)]
RETURN datatype
(IS|AS)
func_body;
```

Functions – Parameters

- Only IN parameters are allowed
    - Use RETURN value instead of OUT
    - If returning multiple values a structure is needed
- You have to define which datatype the function returns

Functions – Example

Example

```
CREATE OR REPLACE FUNCTION max_won
    RETURN matches.won%TYPE
AS
    v_max_won matches.won%TYPE;
BEGIN
    SELECT MAX(won) INTO v_max_won FROM matches;
    RETURN v_max_won;
END;
```

Functions

How to call

```
BEGIN
    dbms_output.put_line(max_won);
END;
```

Retrieve object info

```
DESCRIBE max_won;
```

Package

- A package is a schema object
- It groups logically dependent Types, Variables, Constants, Exceptions and subroutines
- A package consists of
  - A specification defining the interface (comparable to a C header file)
  - A body containing the actual implementation and 'private' members
- Packages can be used by other programs
- The whole package is loaded into memory if any piece of it is referenced

Package – Benefits

- Enclosure of related constructs
- Improvements for the design by separating specification and body
- Hiding 'private' methods
- Allows to persist variables and cursors for the duration of a session ← yet we do not want to rely on that
- Two edged blade: performance
  - The whole package is loaded into memory with the first usage
  - That speeds up subsequent accesses
  - But it also wastes a lot of memory if only a fraction is actually needed

HTL LEONDING

Package – Specification

Syntax

```
CREATE [OR REPLACE] PACKAGE packageName
(IS | AS)
    <public types and variables>
    <public subroutine prototypes>
END [packageName];
```

- Represents the interface used by other applications

Package – Body

Syntax

```
CREATE [OR REPLACE] PACKAGE BODY packageName
(IS | AS)
    <private types and variables>
    <public and private subroutine implementations>
[BEGIN <initialization statements>]
END [packageName];
```

- Contains the implementation of the public subroutines and the supporting private subroutines
- BEGIN Block will be executed when the package is referenced for the first time (cf. constructor)

Package – Example – Specification

Specification

```
CREATE OR REPLACE PACKAGE circle_funcs
AS
    c_pi CONSTANT NUMBER(3,2) := 3.14;

    FUNCTION circumference(p_radius NUMBER) RETURN
        NUMBER;
    FUNCTION area(p_radius NUMBER) RETURN NUMBER;
END circle_funcs;
```

Package – Example – Body

Body

```
CREATE OR REPLACE PACKAGE BODY circle_funcs
AS
    FUNCTION circumference(p_radius NUMBER) RETURN NUMBER AS
        BEGIN
            RETURN p_radius * 2.0 * c_pi;
        END;
    FUNCTION area(p_radius NUMBER) RETURN NUMBER AS
        BEGIN
            RETURN p_radius * p_radius * c_pi;
        END;
END;
```

HTL LEONDING

Package – Example – Usage

Usage

```
SET SERVEROUTPUT ON;
SELECT circle_funcs.circumference(2) FROM dual;
/
BEGIN
    dbms_output.put_line(circle_funcs.circumference(2));
    dbms_output.put_line(circle_funcs.area(2));
END;
```

Trigger

- A trigger is a code block stored in the database
- Similar to functions and procedures
- Yet it is activated (*triggered*) automatically
- Events that can activate a trigger:
  - DML-Statements (INSERT, UPDATE, DELETE)
  - DDL-Statements (CREATE, ALTER, DROP)
  - Database Events (LOGON, STARTUP, SHUTDOWN,. . .)

Trigger – Scenarios

- The following scenarios might warrant the use of a trigger:
  - Security
  - Auditing
  - Data Integrity
  - Referential Integrity
  - Table replication
  - Calculating derived data
  - Event protocol

Trigger – Management

- (De)activating a trigger:
  - **ALTER TRIGGER** trigger_name ENABLE | DISABLE
- Deleting a trigger:
  - **DROP TRIGGER** trigger_name
- Querying information about a trigger:
  - **SELECT** * **FROM** USER_TRIGGERS **WHERE** Trigger_name = 'name'

HTL LEONDING

DML-Trigger

Syntax

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT | DELETE | UPDATE [OF col1[,col2,...]]}
[OR {INSERT | DELETE | UPDATE ...}] ...
ON object_name
[REFERENCES [OLD AS old] [NEW AS new]]
[FOR EACH ROW [WHEN condition]]
<PL/SQL−Block>
```

DML-Trigger – Example

Example

```
CREATE TRIGGER audit_emp_sal
    AFTER UPDATE OF sal ON emp
    FOR EACH ROW
BEGIN
    INSERT INTO emp_audit VALUES ...
END;
```

DML-Trigger – Example

- **UPDATE** [OF col1]
  - Trigger can be restricted to check for updates on specific columns
- **FOR** EACH ROW [**WHEN** condition]
  - Trigger runs for each row fulfilling the condition
- The keywords :new & :old can be used to access the original and the updated value
  - REFERENCES can be used to overwrite those values
- In the PL/SQL block the triggering event can be determined:
  - **IF** INSERTING **THEN**
  - **IF** UPDATING **THEN**
  - **IF** DELETING **THEN**

INSTEAD OF Trigger

- Instead of a specific action the trigger action runs
- Often used when attempting to insert into a read-only view

Example

```
CREATE OR REPLACE TRIGGER emp_view_insert
    INSTEAD OF INSERT on emp_dep_view
    FOR EACH ROW
    BEGIN
        INSERT INTO emp VALUES ...;
        ...
    END;
```

DDL-Trigger

- Can be declared on database and on schema level
- Examples for triggering event: CREATE, DROP, GRANT,. . .

Example

```
CREATE OR REPLACE TRIGGER drop_trigger
    BEFORE DROP ON scott.SCHEMA
    BEGIN
        RAISE_APPLICATION_ERROR(−20000, ’Drop not allowed’);
    END;
```

HTL LEONDING