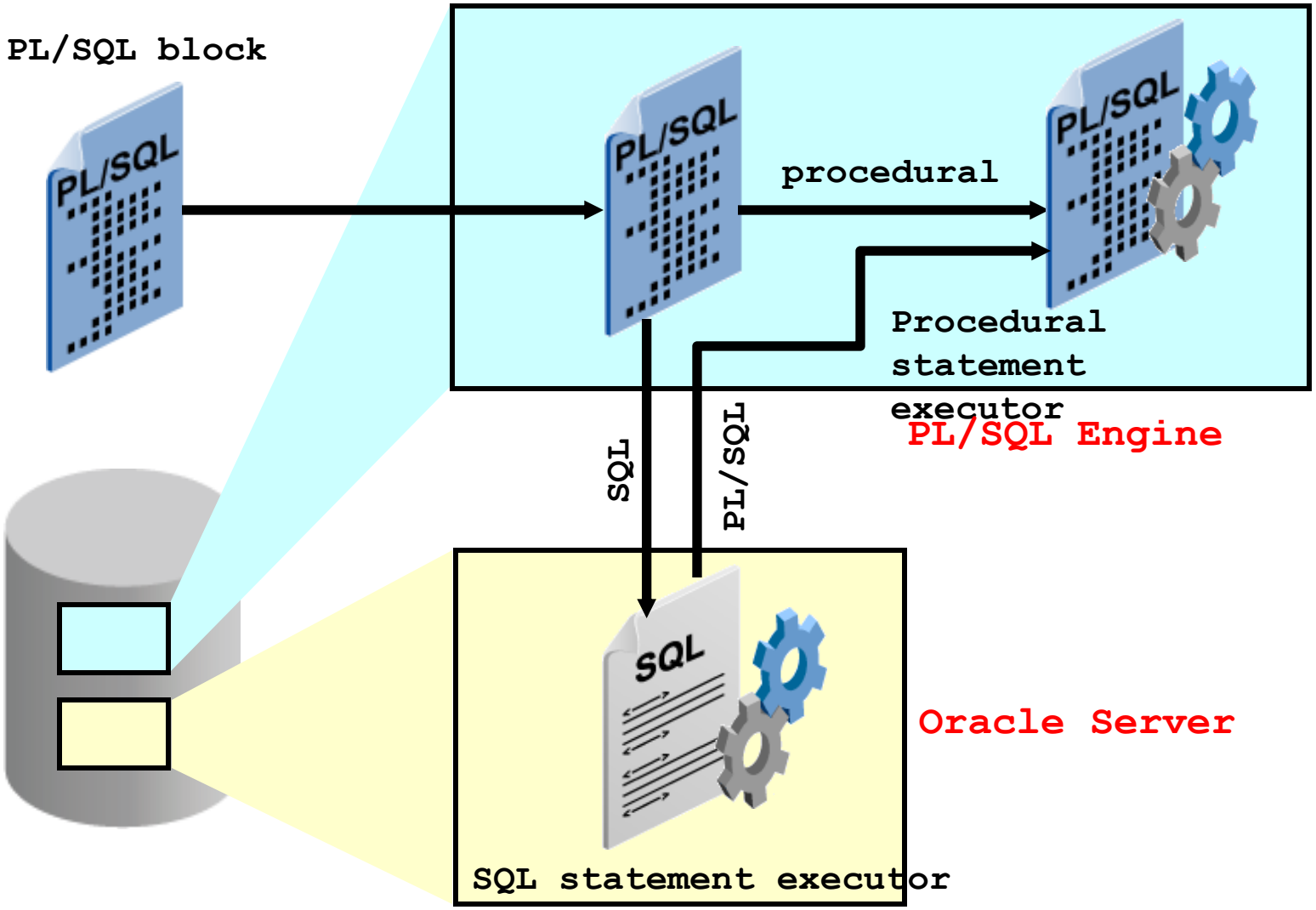
PL/SQL Programming Concepts: Review

PL/SQL Run-Time Architecture



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;
```

PL/SQL Block Structure

- **DECLARE (optional)**
  - Variables, cursors, user-defined exceptions
- **BEGIN (mandatory)**
  - SQL statements
  - PL/SQL statements
- **EXCEPTION (optional)**
  - Actions to perform when exceptions occur
- **END; (mandatory)**

Types of Variables

- **PL/SQL variables:**
  - **Scalar**
  - **Reference**
  - **Large object (`LOB`)**
  - **Composite (Record, Collection)**
- **Non-PL/SQL variables: Bind variables**

Declaring and Initializing PL/SQL Variables

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

Examples:

```
DECLARE
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
  v_salary        NUMBER(4):=4000;
  v_minsalary     v_salary%type;
  v_maxsalary     v_salary%type:=v_salary+6000;
  v_boolean       BOOLEAN;
  v_boolean_init  BOOLEAN NOT NULL DEFAULT=true;
```

Composite Data Types: Records and Collections

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	

PL/SQL Collections:

1	SMITH
2	JONES
3	NANCY
4	TIM

PLS_INTEGER — VARCHAR2

1	5000
2	2345
3	12
4	3456

PLS_INTEGER — NUMBER

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row functions
- Not available in procedural statements:
  - `DECODE`
  - `NVL2`
  - `COALESCE`
  - `NULLIF`
  - Group functions

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations

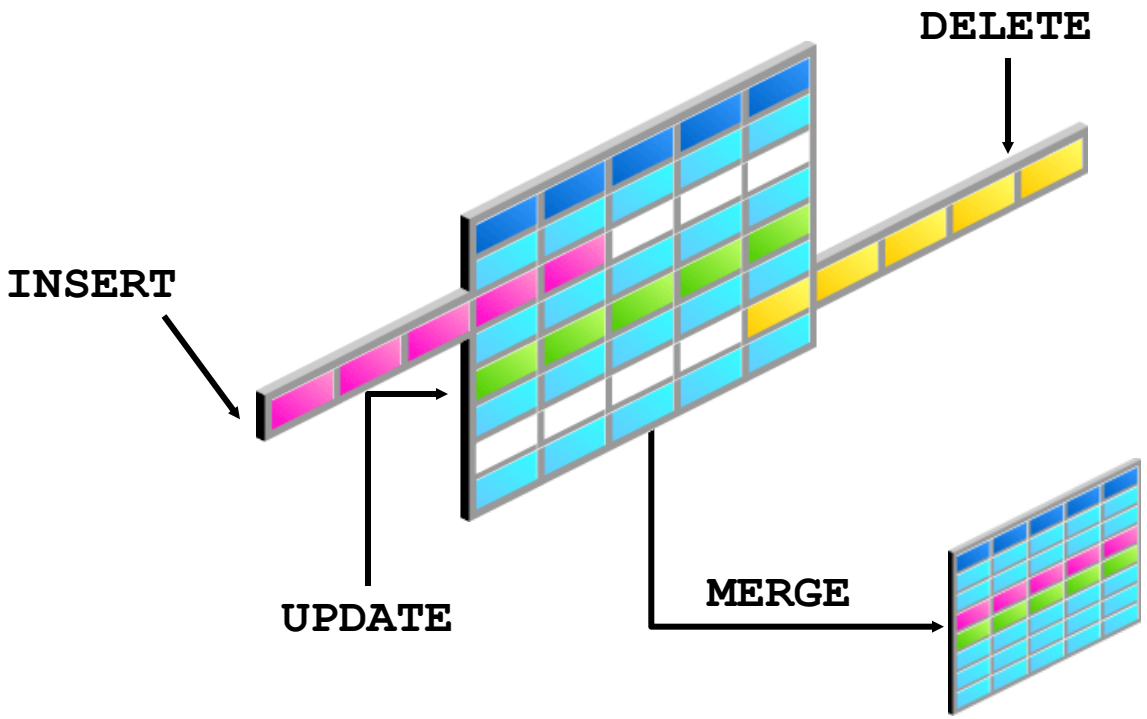
Same as in SQL

- Exponential operator (**)

Using PL/SQL to Manipulate Data

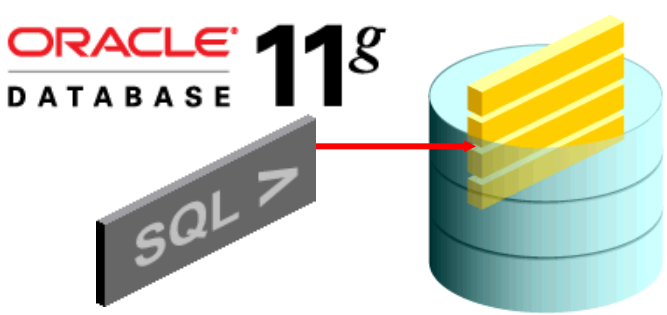
Make changes to database tables by using DML and transactional statements:

- INSERT
- UPDATE
- DELETE
- MERGE
- COMMIT
- ROLLBACK
- SAVEPOINT

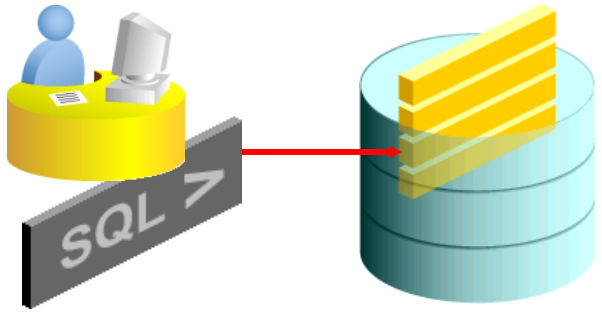


SQL Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle Server. It is used to handle the result set of a `SELECT` statement.
- There are two types of cursors: implicit and explicit.
  - Implicit: Created and managed internally by the Oracle Server to process SQL statements
  - Explicit: Declared explicitly by the programmer



Implicit cursor



Explicit cursor

SQL Cursor Attributes for Implicit Cursors

Using SQL cursor attributes, you can test the outcome of your SQL statements.

SQL%FOUND	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement affected at least one row
SQL%NOTFOUND	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement did not affect even one row
SQL%ROWCOUNT	An integer value that represents the number of rows affected by the most recent SQL statement

SQL Cursor Attributes for Implicit Cursors

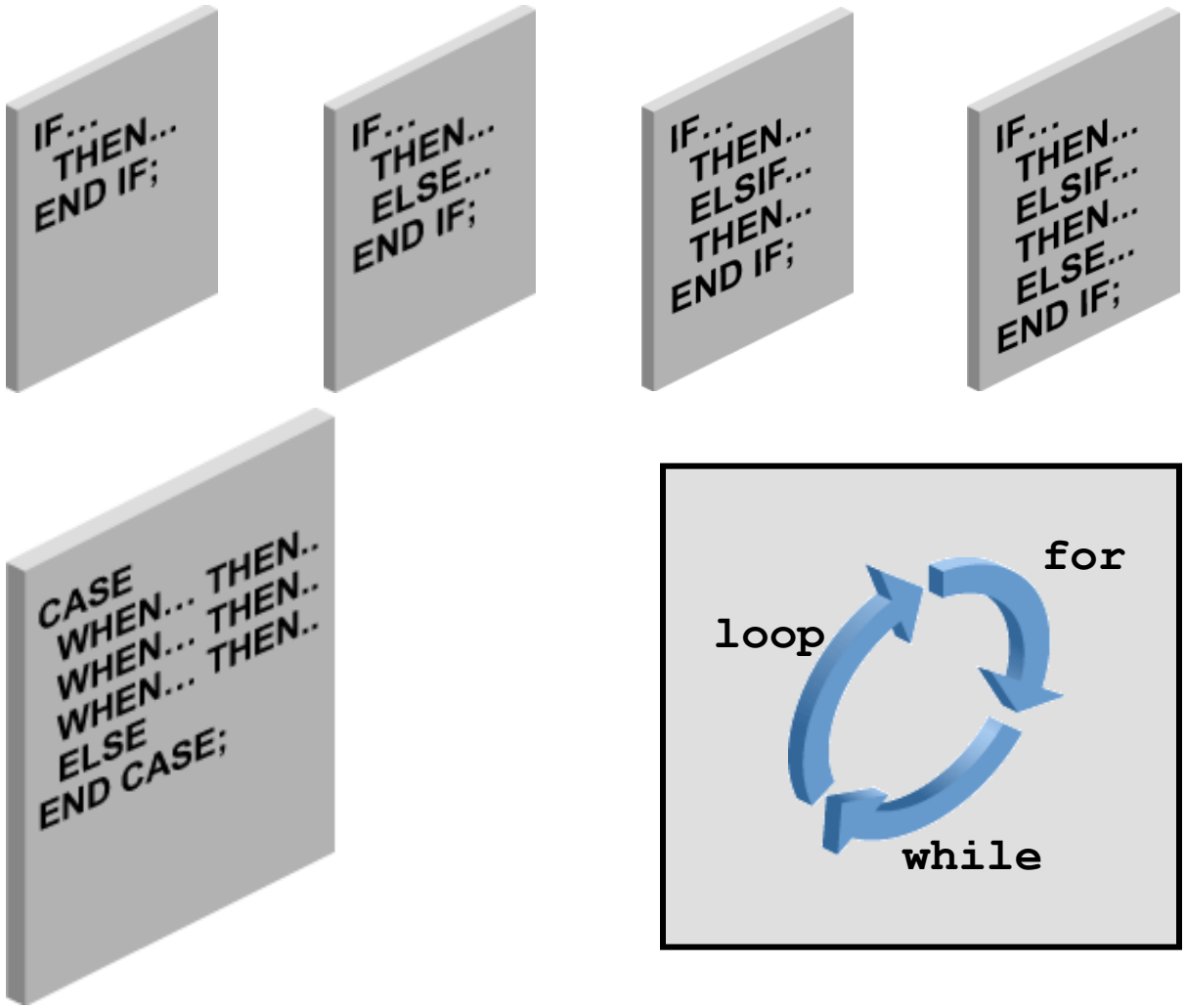
Delete rows that have the specified employee ID from the `employees` table. Print the number of rows deleted.

Example:

```
DECLARE
  v_rows_deleted VARCHAR2(30)
  v_empno employees.employee_id%TYPE := 176;
BEGIN
  DELETE FROM  employees
  WHERE employee_id = v_empno;
  v_rows_deleted := (SQL%ROWCOUNT ||
                        ' row deleted.');
  DBMS_OUTPUT.PUT_LINE (v_rows_deleted);

END;
```

Controlling Flow of Execution



Logic Tables

Build a simple Boolean condition with a comparison operator.

AND	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Example for anonymous PL/SQL block

```
DECLARE
N NUMBER:=&SZAM;
J NUMBER;
BEGIN
<<KULSO>>
LOOP
  FOR I IN 2..SQRT(N) LOOP
  IF N/I=TRUNC(N/I) THEN
   DBMS_OUTPUT.PUT_LINE(N||' NEM PRIM, '||I ||' OSZTJA');
   EXIT KULSO;
  END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(N||' PRIM');
  EXIT;
END LOOP;
END;
/
```

Processing Explicit Cursors

The following three commands are used to process an explicit cursor:

- **`OPEN`**
- **`FETCH`**
- **`CLOSE`**

Alternatively, you can also use a cursor `FOR` loops.

Explicit Cursor Attributes

Every explicit cursor has the following four attributes:

- `cursor_name%FOUND`
- `cursor_name%ISOPEN`
- `cursor_name%NOTFOUND`
- `cursor_name%ROWCOUNT`

Example for explicit cursor

```
DECLARE CURSOR C_EMP IS
SELECT t.*,12*SALARY*(1+NVL(COMMISSION_PCT,0)) EVES_FIZ
FROM EMPLOYEES t
WHERE DEPARTMENT_ID=&OSZTALYKOD;
R C_EMP%ROWTYPE;
SUMMA NUMBER;
BEGIN
SUMMA:=0;
OPEN C_EMP;
LOOP
  FETCH C_EMP INTO R ;
  EXIT WHEN C_EMP%NOTFOUND;
  SUMMA:=SUMMA+R.SALARY;
   DBMS_OUTPUT.PUT_LINE(
   C_EMP%ROWCOUNT||'. DOLGOZO:='||RPAD(R.LAST_NAME,15,' ')
   ||' FIZ:'||R.SALARY||' FONOKE:'||R.MANAGER_ID
   ||' EVES JOV:'||R.EVES_FIZ);
END LOOP;
DBMS_OUTPUT.PUT_LINE(
CHR(10)||C_EMP%ROWCOUNT||' OSSZES FIZETES: '||SUMMA);
CLOSE C_EMP;
END;
/
```

Cursor FOR Loops

Syntax:

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

Cursor with FOR: Example

```
DECLARE
 CURSOR C_DEPT  IS
 SELECT *  FROM departments;
 CURSOR C_EMP (C_DEPTNO NUMBER) IS
 SELECT *  FROM employees
 WHERE DEPARTMENT_ID=C_DEPTNO;
 BEGIN
 DBMS_OUTPUT.ENABLE(1000000);
 FOR R IN C_DEPT LOOP
   DBMS_OUTPUT.PUT_LINE(CHR(10)||R.DEPARTMENT_ID
   ||' '||R.DEPARTMENT_NAME||CHR(10));
   FOR Q IN C_EMP(R.DEPARTMENT_ID) LOOP
     DBMS_OUTPUT.PUT_LINE(C_EMP%ROWCOUNT||'. DOLGOZO:'||
     Q.EMPLOYEE_ID||' '||Q.LAST_NAME);
   END LOOP;
 END LOOP;
 END;
/
```

Handling Exceptions

- An exception is an error in PL/SQL that is raised during program execution.
- An exception can be raised:
  - Implicitly by the Oracle server
  - Explicitly by the program
- An exception can be handled:
  - By trapping it with a handler
  - By propagating it to the calling environment

Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND (SELECT)
  - TOO_MANY_ROWS (SELECT)
  - INVALID_CURSOR (FECTH from closed cursor)
  - ZERO_DIVIDE (1/0 de nem 1F/0 !)
  - DUP_VAL_ON_INDEX (INSERT or UPDATE)
  - VALUE_ERROR (too large data
  - SUBSCRIPT_BEYOND_COUNT (index is over for collection)

```
SELECT text
FROM all_source
WHERE  name='STANDARD' AND UPPER(text) LIKE UPPER('%&KERES%')
/
```

Example of exception handling
Predefined exception

```
DECLARE
er employees%ROWTYPE;
BEGIN
SELECT * INTO er
FROM employees
WHERE employee_id=&DKOD;
DBMS_OUTPUT.PUT_LINE
(er.first_name||' ' ||er.last_name||' fizetese:'||er.salary);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('Nincs ilyen dolgozo!');
END;
/
```

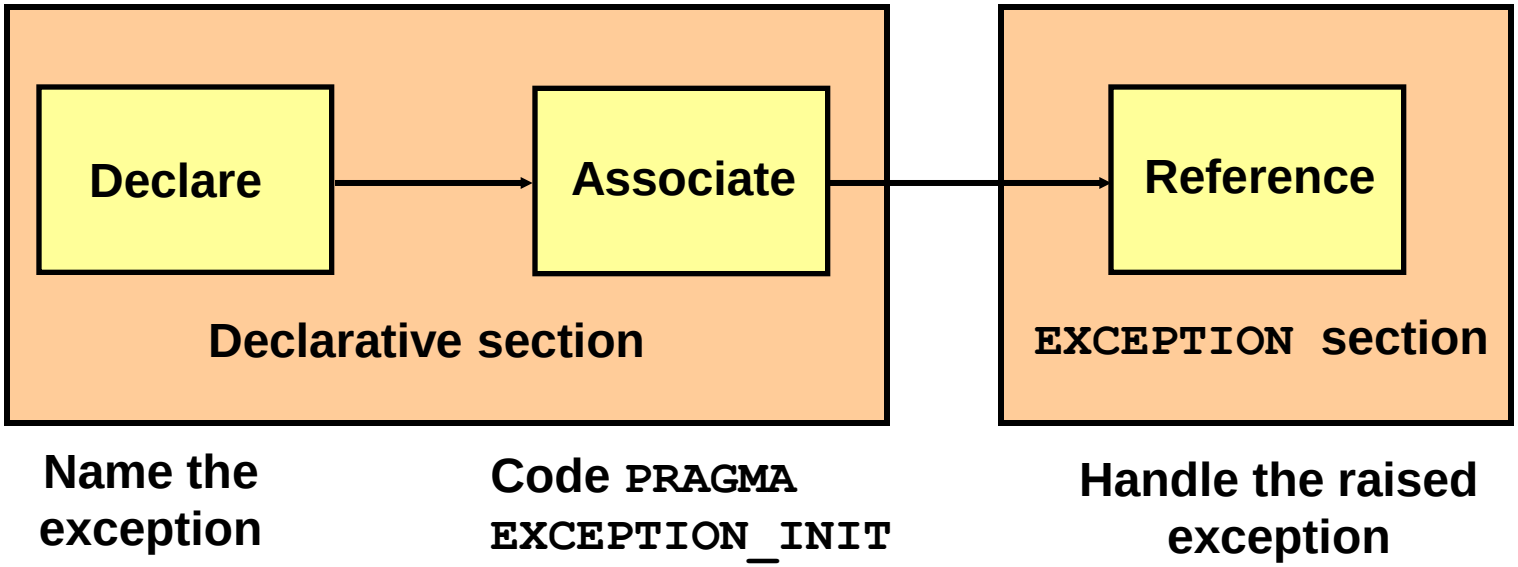
Handling Exceptions: Bad example

```
DECLARE
w employees%ROWTYPE;
m employees%ROWTYPE;
d departments%ROWTYPE;
BEGIN
SELECT * INTO w FROM employees   WHERE employee_id  =&empno;
SELECT * INTO m FROM employees   WHERE employee_id  =w.manager_id;
SELECT * INTO d FROM departments WHERE department_id=w.department_id;
DBMS_OUTPUT.PUT_LINE
(w.last_name||','||m.last_name||','||d.department_name);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('The error was: '||SQLERRM);
END;
/
```

Handling Exceptions: Good example

```
DECLARE w employees%ROWTYPE; m employees%ROWTYPE; d departments%ROWTYPE;
BEGIN
  BEGIN
    SELECT * INTO w FROM employees WHERE employee_id=&empno;
    EXCEPTION WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such an employee'); RAISE;
  END;
  BEGIN
    SELECT * INTO m FROM employees WHERE employee_id=w.manager_id;
    EXCEPTION WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No manager!');
  END;
  BEGIN
    SELECT * INTO d FROM departments
    WHERE department_id=w.department_id;
    EXCEPTION WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No department!');
  END;
DBMS_OUTPUT.PUT_LINE
(w.last_name||','||m.last_name||','||d.department_name);
EXCEPTION WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('The error was: '||SQLERRM);
END;
/
```

Trapping Non-Predefined Oracle Server Errors



Example of exception handling Non-Predefined exception

```
DECLARE
  nincs_ilyen_dolgozo EXCEPTION ;
  nincs_ilyen_osztaly EXCEPTION ;
PRAGMA EXCEPTION_INIT (nincs_ilyen_osztaly, -2291) ;
BEGIN
 UPDATE employees SET department_id = &OSZTALY
 WHERE employee_id = &DOLGOZO;
 IF SQL%NOTFOUND THEN
        RAISE nincs_ilyen_dolgozo;
 END IF ;
DBMS_OUTPUT.PUT_LINE('SIKERULT !') ;
EXCEPTION
 WHEN nincs_ilyen_dolgozo THEN
   DBMS_OUTPUT.PUT_LINE('Nincs ilyen dolgozo !') ;
 WHEN nincs_ilyen_osztaly THEN
   DBMS_OUTPUT.PUT_LINE('Nincs ilyen osztaly !') ;
END;
/
```

The RAISE_APPLICATION_ERROR Procedure

Syntax:

```
raise_application_error (error_number,
                         message[, {TRUE | FALSE}]);
```

- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

Procedures

A procedure is:

- A named PL/SQL block that performs a sequence of actions
- Stored in the database as a schema object
- Used to promote reusability and maintainability

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; …]
BEGIN
  -- actions;
END [procedure_name];
```

Procedure: Example

```
CREATE OR REPLACE PROCEDURE osztalyok
(p_deptno employees.department_id%TYPE DEFAULT 90)
IS
CURSOR C_EMP(c_deptno employees.department_id%TYPE) IS
SELECT t.*,12*salary*(1+NVL(commission_pct,0)) ANN_SAL
FROM employees t
WHERE DEPARTMENT_ID=c_deptno;
s NUMBER:=0; MANAGER_NAME EMPLOYEES.LAST_NAME%TYPE;
BEGIN
FOR R IN C_EMP(p_deptno) LOOP
  s:=s+R.salary;
  IF R.MANAGER_ID IS NOT NULL THEN
  SELECT LAST_NAME INTO MANAGER_NAME
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID=R.MANAGER_ID;
  ELSE   MANAGER_NAME:='Nincs';
  END IF;
  DBMS_OUTPUT.PUT_LINE(R.LAST_NAME||' SALARY:'||R.salary||
  ' MANAGER:'||MANAGER_NAME||' ANNUAL_SALARY:'||R.ANN_SAL);
END LOOP;
DBMS_OUTPUT.PUT_LINE(CHR(10)||' TOTAL SALARIES: '||s);
END OSZTALYOK;
/
```

Functions

A function is:

- A block that returns a value
- Stored in the database as a schema object
- Called as part of an expression or used to provide a parameter value

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, ...)]
RETURN datatype IS|AS
 [local_variable_declarations; …]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

Function: Example

```
CREATE OR REPLACE FUNCTION osztaly_fiz
(P_DEPTNO EMPLOYEES.DEPARTMENT_ID%TYPE:=10)
RETURN NUMBER
IS
SUMMA NUMBER;
BEGIN
SELECT SUM(SALARY) INTO SUMMA FROM EMPLOYEES
WHERE DEPARTMENT_ID=P_DEPTNO;
IF SUMMA IS NULL THEN
  RETURN -1;
ELSE
  RETURN SUMMA;
END IF;
END OSZTALY_FIZ;
/
```

```
EXECUTE dbms_output.put_line(osztaly_fiz(90))
SELECT d.*,osztaly_fiz(department_id)
FROM departments d;
```

Restrictions on Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  - Be stored in the database
  - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
  - Return valid SQL data types, not PL/SQL-specific types
  - Parameters must be specified with positional notation
  - You must own the function or have the `EXECUTE` privilege
  - A `SELECT` statement cannot contain DML statements
  - An `UPDATE` or `DELETE` statement on a table `T` cannot query or contain DML on the same table `T`
  - SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)

Generating PI

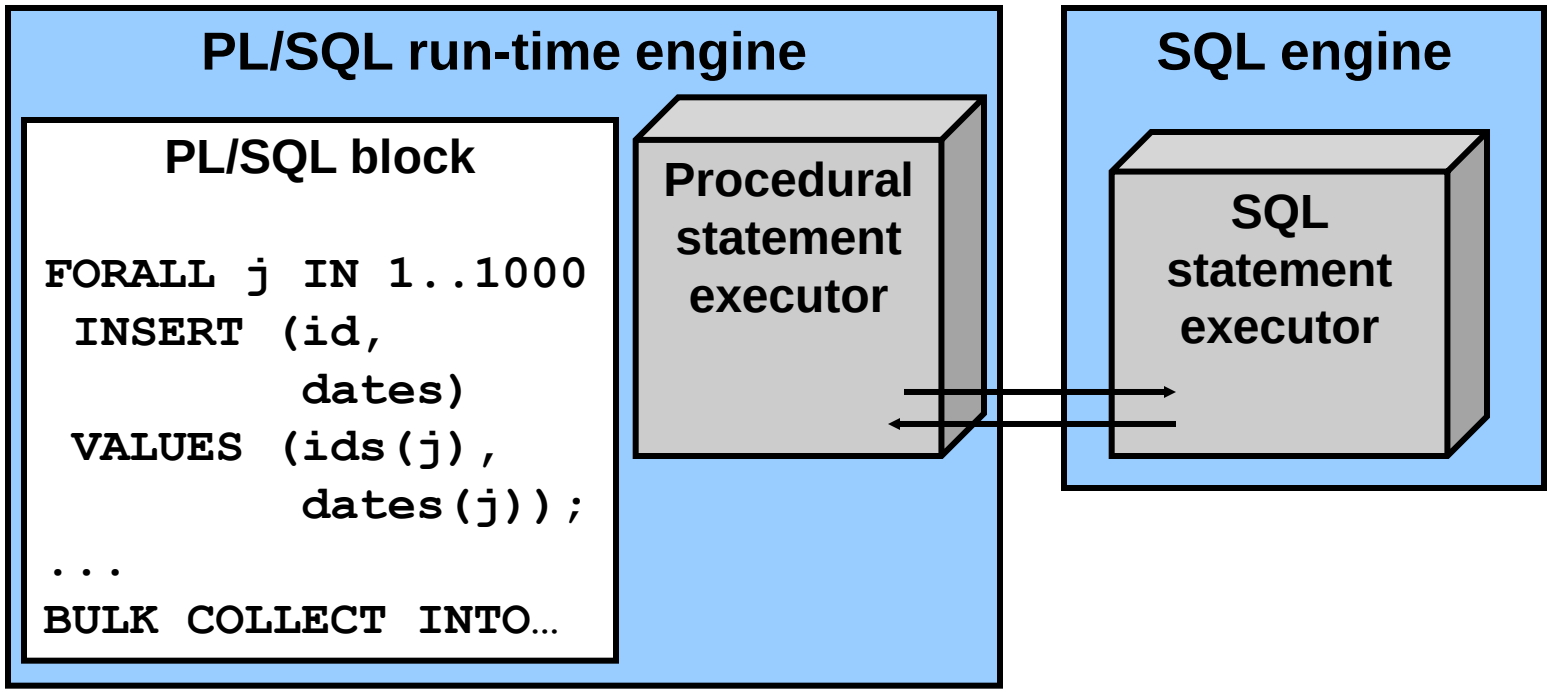
```
CREATE OR REPLACE FUNCTION PI(EPS NUMBER:=1E-15)
return BINARY_DOUBLE --Newton method
IS
PREVIOUS BINARY_DOUBLE:=0D;
ACTUAL BINARY_DOUBLE:=0.5D;
N NUMBER:=1D; K NUMBER:=2D;
F BINARY_DOUBLE:=1D;
I PLS_INTEGER:=0;
BEGIN
WHILE ABS(ACTUAL-PREVIOUS)>EPS LOOP
I:=I+1;
PREVIOUS:=ACTUAL;
F:=F*N/K;
N:=N+2.0; K:=K+2.0;
ACTUAL:=PREVIOUS+F*(0.5D**N)/N;
END LOOP;
RETURN 6D*ACTUAL;
END;
/
SELECT TO_CHAR(pi, '9.999999999999') FROM dual;
```

Native dynamic SQL
DDL statement in PL/SQL

```
CREATE OR REPLACE PROCEDURE cre_tab
(p_tab_name IN  VARCHAR2 DEFAULT 'emp_temp',
 p_from     IN  VARCHAR2 DEFAULT 'employees')
 IS
 stmt VARCHAR2(2000);
 letezo_objektum EXCEPTION;
 PRAGMA EXCEPTION_INIT(letezo_objektum,-942);
 BEGIN
 stmt:=' CREATE TABLE '||p_tab_name ||
       ' AS SELECT * FROM '||p_from;
   EXECUTE IMMEDIATE stmt;
 EXCEPTION
 WHEN letezo_objektum THEN
 DBMS_OUTPUT.PUT_LINE('Ilyen objektum mar van');
 END;
/
```

Bulk Binding

Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



Compare the normal and the Bulk Binding

```
DROP TABLE PARTS; CREATE TABLE parts(n number, t varchar2(100));
DECLARE   TYPE NumTab IS TABLE OF NUMBER(15) INDEX BY
    BINARY_INTEGER;
TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY_INTEGER;
   pnums  NumTab;pnames NameTab; n1 number; n2 number;
   BEGIN
    FOR j IN 1..50000 LOOP  -- load index-by tables
      pnums(j) := j; pnames(j) := 'Part No.'||TO_CHAR(j);
    END LOOP;
    n1:=dbms_utility.get_cpu_time;
    FOR i IN 1..50000 LOOP  -- use FOR loop
       INSERT INTO parts VALUES (pnums(i), pnames(i));
    END LOOP;
    n2:=dbms_utility.get_cpu_time;
     DBMS_OUTPUT.PUT_LINE('diff :'||to_char((n2-n1)/100));
    n1:=dbms_utility.get_cpu_time;
    FORALL i IN 1..50000  -- use FORALL statement
    INSERT INTO parts VALUES (pnums(i), pnames(i));
    n2:=dbms_utility.get_cpu_time;
     DBMS_OUTPUT.PUT_LINE('diff2:'||to_char((n2-n1)/100));
 END;
```

Working with traditional FETCH

```
CREATE TABLE BIG_EMP(EMPNO,LAST_NAME,FIRST_NAME,SALARY,DEPARTMENT_ID)
AS
SELECT E.EMPLOYEE_ID||ROWNUM,E.LAST_NAME||ROWNUM,E.FIRST_NAME||ROWNUM,
E.SALARY,E.DEPARTMENT_ID
FROM EMPLOYEES E,EMPLOYEES D,EMPLOYEES F;

CREATE OR REPLACE PROCEDURE  trad_fetch IS
CURSOR c_big_emp is SELECT * FROM big_emp;
S NUMBER:=0;  n1 number;   n2 number;
EMP2 BIG_EMP%ROWTYPE;
BEGIN
n1:=dbms_utility.get_cpu_time;
OPEN c_big_emp;
   LOOP
      FETCH c_big_emp INTO EMP2 ;
      EXIT WHEN c_big_emp %NOTFOUND ;
      S:=S+ EMP2.SALARY;
END LOOP;
CLOSE c_big_emp;
n2:=dbms_utility.get_cpu_time;
 DBMS_OUTPUT.PUT_LINE('diff:'||to_char((n2-n1)/100)||'S:='||S);
END;
/
EXEC trad_fetch
```

BULK BINDING for FETCH statement

```
CREATE OR REPLACE PROCEDURE BULK_LIMIT(rows NUMBER := 10)
IS
CURSOR c_big_emp is SELECT * FROM big_emp;
type c_type is table of  BIG_EMP%rowtype;
emp c_type; j number:=0;
S NUMBER:=0;     n1 number;   n2 number; stmt varchar2(200);
BEGIN
n1:=dbms_utility.get_cpu_time;
 OPEN c_big_emp;
 LOOP
      FETCH c_big_emp BULK COLLECT INTO EMP LIMIT rows;
      EXIT WHEN c_big_emp%NOTFOUND and emp.count=0;
      FOR I IN 1..EMP.COUNT LOOP
      S:=S+ EMP(I).SALARY; END LOOP;
   END LOOP;
 CLOSE c_big_emp ;
n2:=dbms_utility.get_cpu_time;
DBMS_OUTPUT.PUT_LINE('diff:'||to_char((n2-n1)/100)||'S:='||S);
END;
/
EXEC BULK_LIMIT(1000)
```

Native dynamic SQL with Bulk Binding

```
CREATE OR REPLACE PROCEDURE FETCH_LIMIT
  (tname VARCHAR2,rows NUMBER DEFAULT 10)
   IS
   TYPE emp_cur_type IS REF CURSOR;
   TYPE c_type is table of  employees%ROWTYPE;
   emp c_type;
   c1 emp_cur_type;
   j number:=0;
BEGIN
  OPEN c1 FOR 'SELECT * FROM '||tname;
  LOOP
  j:=j+1;
  dbms_output.put_line('NUMBER OF LOOPS:'||j);
  FETCH c1 BULK COLLECT INTO emp LIMIT rows;
  FORALL I IN emp.FIRST..emp.LAST
  INSERT INTO NEWEMP VALUES emp(I);
  EXIT WHEN c1%NOTFOUND;
  END LOOP;
  CLOSE c1;
END FETCH_LIMIT; /* DROP TABLE NEWEMP; */
/
```

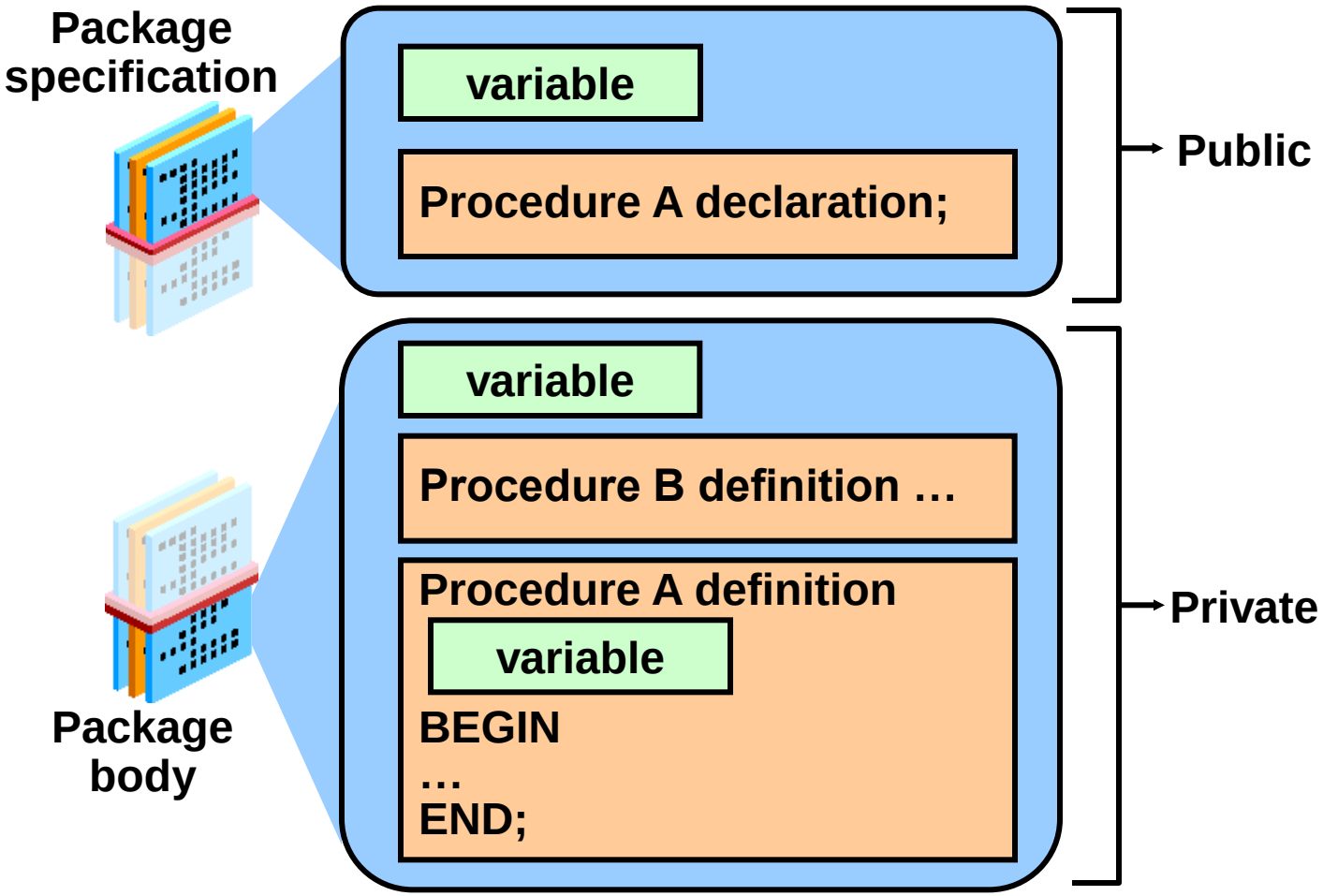
PL/SQL Packages: Review

PL/SQL packages:

- **Group logically related components:**
  - **PL/SQL types**
  - **Variables, data structures, and exceptions**
  - **Subprograms: procedures and functions**
- **Consist of two parts:**
  - **A specification**
  - **A body**

- **Enable the Oracle server to read multiple objects into memory simultaneously**

Components of a PL/SQL Package



Creating the Package Specification

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name IS|AS
    public type and variable declarations
    subprogram specifications
END [package_name];
```

- The `OR REPLACE` option drops and re-creates the package specification.
- Variables declared in the package specification are initialized to `NULL` by default.
- All the constructs declared in a package specification are visible to users who are granted privileges on the package.

Creating the Package Body

Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS|AS
    private type and variable declarations
    subprogram bodies
[BEGIN initialization statements]
END [package_name];
```

- The `OR REPLACE` option drops and re-creates the package body.
- Identifiers defined in the package body are private and not visible outside the package body.
- All private constructs must be declared before they are referenced.
- Public constructs are visible to the package body.

Package : Example

```
CREATE OR REPLACE PACKAGE cs
IS
CURSOR c_emp(c_deptno employees.department_id%TYPE) IS
SELECT employee_id,last_name,salary,manager_id
FROM employees
WHERE department_id=c_deptno;
v_sal NUMBER:=11;
no_parent EXCEPTION;
PRAGMA EXCEPTION_INIT(NO_PARENT,-2291);
PROCEDURE print ( what VARCHAR2 );
END cs;
/
CREATE OR REPLACE PACKAGE BODY cs IS
 PROCEDURE print ( what VARCHAR2 ) IS
BEGIN
     DBMS_OUTPUT.PUT_LINE(what);
END print;
END cs;
/
```

Standardize everything!
(exceptions,cursors,variables,types etc)

```
CREATE OR REPLACE PACKAGE exceptions IS
  no_parent   EXCEPTION; PRAGMA EXCEPTION_INIT(no_parent   , -2291);
  child_found EXCEPTION; PRAGMA EXCEPTION_INIT(child_found , -2292);
  nowait_exc  EXCEPTION; PRAGMA EXCEPTION_INIT(nowait_exc  ,   -54);
  wait_exc    EXCEPTION; PRAGMA EXCEPTION_INIT(wait_exc    ,-30006);
  no_updated_rows EXCEPTION;
END exceptions;
/
CREATE OR REPLACE PACKAGE cursors IS
CURSOR c_dept(c_deptno employees.department_id%TYPE) IS
       SELECT employee_id,last_name,salary,manager_ID FROM employees
       WHERE department_id=c_deptno;
CURSOR c_job (c_jobid employees.job_id%TYPE) IS
       SELECT employee_id,last_name,salary,manager_ID FROM employees
       WHERE job_id=c_jobid;
END cursors;
/
```

Referring package defined Exceptions

```
BEGIN
  UPDATE employees SET department_id = &P_DEPTNO
  WHERE employee_id = &P_EMPNO  ;
  IF SQL%NOTFOUND THEN
  RAISE exceptions.no_updated_rows ;
  END IF ;
DBMS_OUTPUT.PUT_LINE('ok !') ;
EXCEPTION
  WHEN exceptions.no_updated_rows THEN
    DBMS_OUTPUT.PUT_LINE('No such an employee!') ;
  WHEN exceptions.no_parent THEN
    DBMS_OUTPUT.PUT_LINE('No such a department!') ;
END;
/
```

Overloading

```
CREATE OR REPLACE PACKAGE OVER_LOAD IS
 PROCEDURE PRT ( V_STRING VARCHAR2 );
 PROCEDURE PRT ( V_DATE DATE );
 PROCEDURE PRT ( V_NUMBER NUMBER );
END OVER_LOAD;
/
CREATE OR REPLACE PACKAGE BODY OVER_LOAD IS
PROCEDURE PRT( V_STRING VARCHAR2 ) IS
BEGIN
     cs.ki('THE  STRING: '||V_STRING);
END PRT;
PROCEDURE PRT( V_NUMBER NUMBER ) IS
BEGIN
     cs.ki('THE NUMBER: '||V_NUMBER);
END PRT;
PROCEDURE PRT ( V_DATE DATE ) IS
BEGIN
cs.ki('THE DATE :'||TO_CHAR(V_DATE,'YYYY.MM.DD HH24:MI:SS'));
END PRT;
END OVER_LOAD;
/
exec over_load.prt('12')
exec over_load.prt(12)
exec over_load.prt('02-may-2007')
```

Types of Triggers

A trigger:

- Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or database
- Executes implicitly whenever a particular event takes place
- Can be either of the following:
  - Application trigger: Fires whenever an event occurs with a particular application
  - Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database

Creating DML Triggers

Create DML statement or row type triggers by using:

```
CREATE [OR REPLACE] TRIGGER trigger_name
 timing
 event1 [OR event2 OR event3]
ON object_name
[[REFERENCING OLD AS old | NEW AS new]
 FOR EACH ROW
 [WHEN (condition)]]
trigger_body
```

- **A statement trigger fires once for a DML statement.**
- **A row trigger fires once for each row affected.**

Note: Trigger names must be unique with respect to other triggers in the same schema.

Row level trigger for multi purposes

```
CREATE OR REPLACE TRIGGER EMP_TR BEFORE update OR INSERT OR DELETE
  ON employees  FOR EACH ROW
  DECLARE
  DML CHAR(1); s VARCHAR2(200);MANAGER_SAL NUMBER;
  BEGIN
  IF INSERTING THEN
    SELECT SALARY INTO MANAGER_SAL
    FROM EMPLOYEES  WHERE EMPLOYEE_ID=:NEW.MANAGER_ID;
    IF :NEW.SALARY>MANAGER_SAL THEN
      RAISE_APPLICATION_ERROR(-20555,'Tul nagy fizetes!');
    END IF; DML:='I';
    S:='New Name: '||:NEW.LAST_NAME||' Salary: '||:NEW.SALARY;
  ELSIF UPDATING THEN
    IF :NEW.SALARY<:OLD.SALARY THEN
      RAISE_APPLICATION_ERROR(-20123,'A fizetes nem csokkenhet!');
    END IF;
    DML:='U';
    S:=    'Old Name:'||:OLD.LAST_NAME||' Salary:'||:OLD.SALARY;
    S:=S||' New Name:'||:NEW.LAST_NAME||' Salary:'||:NEW.SALARY;
  ELSE DML:='D';
    S:=    'Old Name:'||:OLD.LAST_NAME||' Salary:'||:OLD.SALARY;
  END IF;
  INSERT INTO HISTORY VALUES(USER,SYSDATE,DML,S);
end EMP_TR;
```