PL/SQL Development

Agenda

- Introduction to PL/SQL
- Declaring PL/SQL Variables
- Writing Executable Statements
- Interacting with the Oracle Server
- Writing Control Structures
- Working with Composite Data Types
- Using Explicit Cursors
- Handling Exceptions
- Creating Stored Procedures and Functions

Agenda

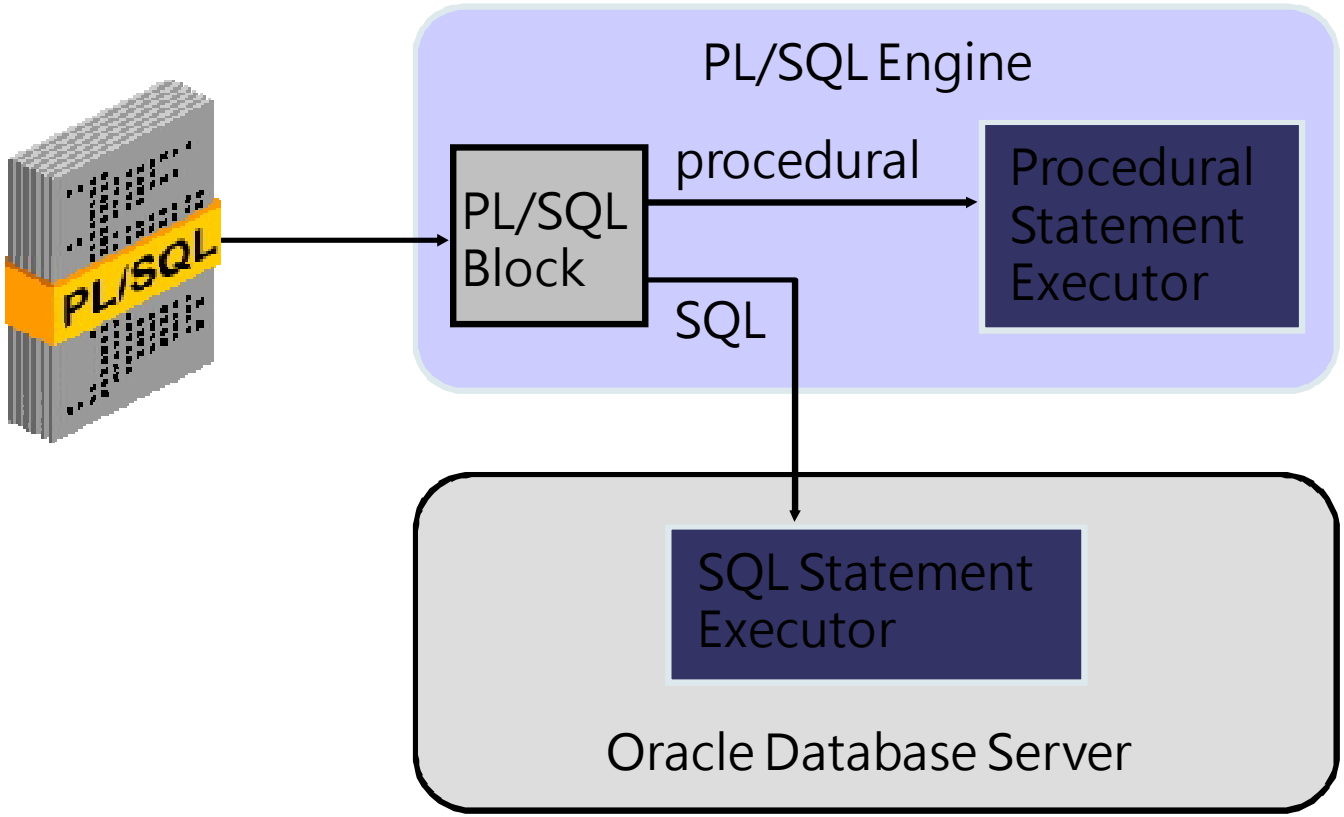
- USER DEFINED TYPE
- DBMS_LOB
- TRIGGER
- DBMS_SCHEDULER
- Lock

Introduction to PL/SQL

What Is PL/SQL?

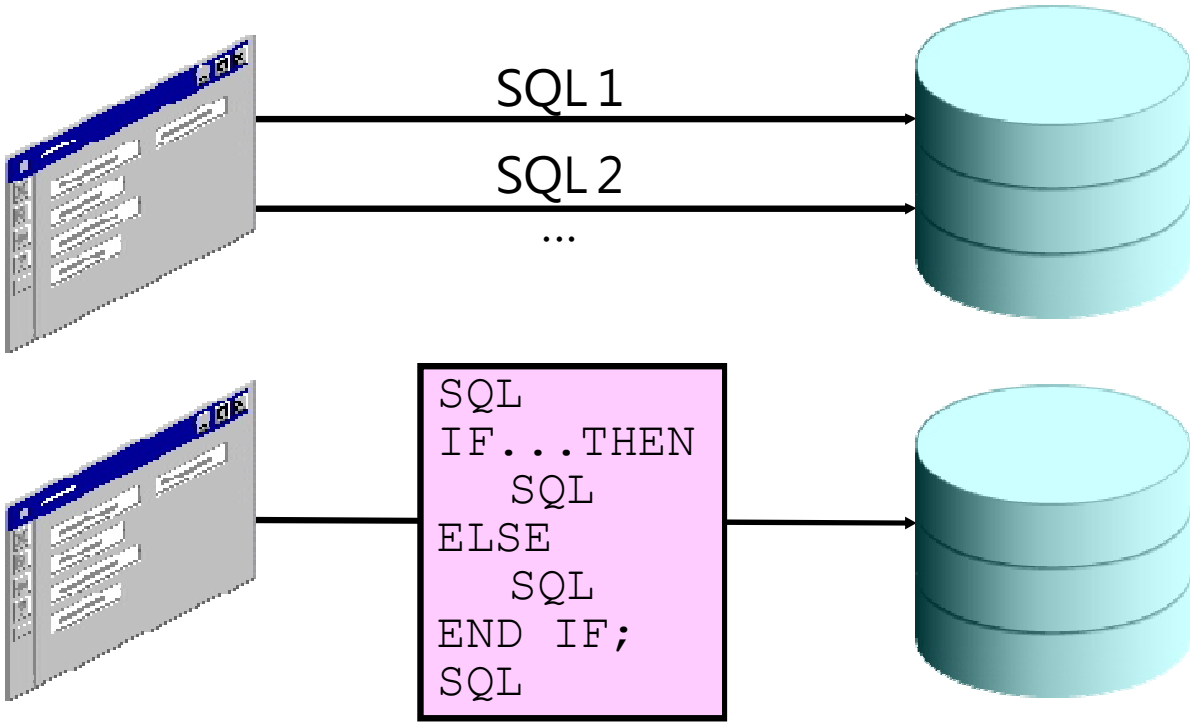
- PL/SQL:
  - Stands for Procedural Language extension to SQL
  - Is Oracle Corporation's standard data access language for relational databases
  - Seamlessly integrates procedural constructs with SQL

PL/SQL Environment



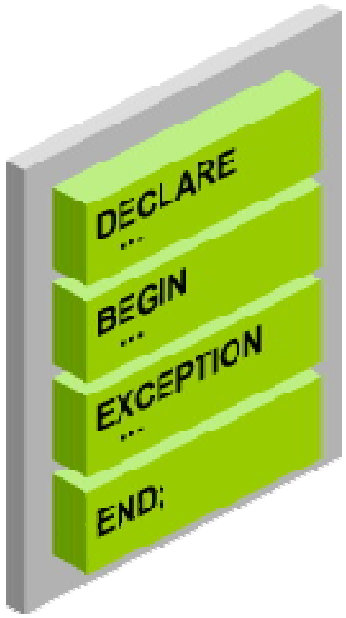
Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance



PL/SQL Block Structure

- `DECLARE` (Optional)
  Variables, cursors, user-defined exceptions
- `BEGIN` (Mandatory)
  - SQL statements
  - PL/SQL statements
- `EXCEPTION` (Optional)
  Actions to perform when errors occur
- `END;` (Mandatory)



Block Types

- Anonymous

```
[DECLARE]


BEGIN
  --statements

[EXCEPTION]

END;
```

Function

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Procedure

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

END;
```

Create an Anonymous Block

- Type the anonymous :

```
DECLARE
        F_name VARCHAR(20);

BEGIN

        SELECT first_name INTO f_name FROM employees WHERE employee_id=100;

END;
```

Test the Output of a PL/SQL Block

- Enable output
  `SET SERVEROUTPUT ON`
- Use a predefined Oracle package and its procedure:
  - `DBMS_OUTPUT.PUT_LINE`

```
SET SERVEROUTPUT ON
…
DBMS_OUTPUT.PUT_LINE(' The First Name of the
Employee is ' || f_name);
…
```

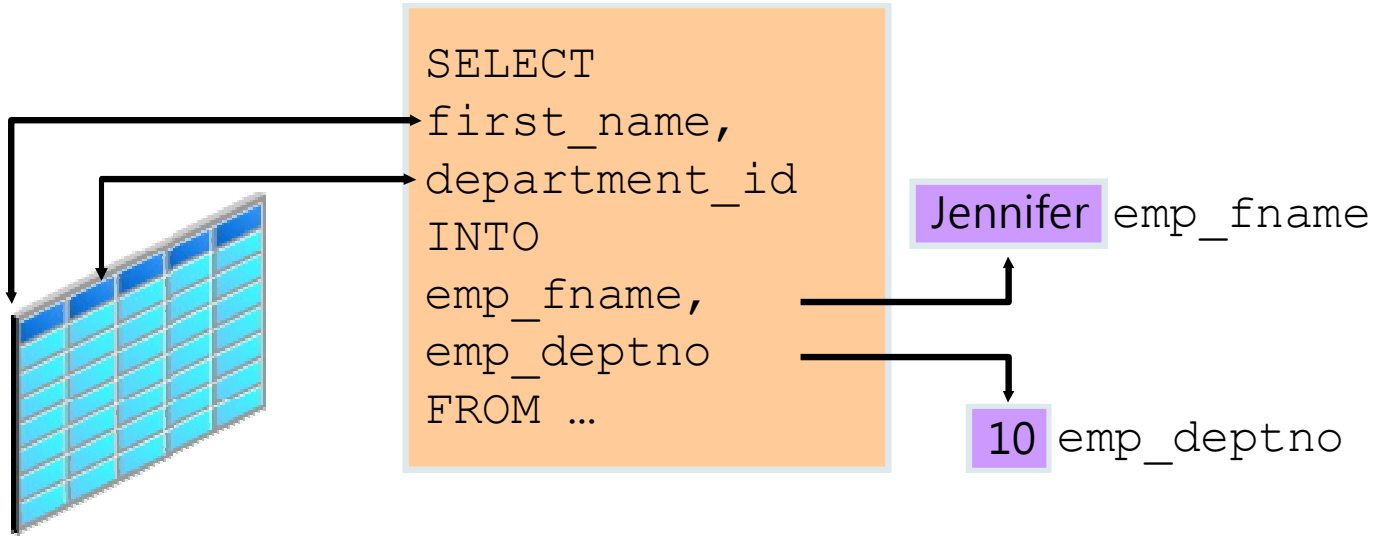
Test the Output of a PL/SQL Block

```
SET SERVEROUTPUT ON

DECLARE
  F_name VARCHAR(20);

BEGIN

  SELECT first_name INTO f_name FROM employees WHERE
employee_id=100;
  DBMS_OUTPUT.PUT_LINE(' The First Name of the Employee is '
|| f_name);

END;
```

Declaring PL/SQL Variables

Use of Variables

- Variables can be used for:
  - Temporary storage of data
  - Manipulation of stored values
  - Reusability



Identifiers

- Identifiers are used for:
  - Naming a variable
  - Providing a convention for variable names:
    - Must start with a letter
    - Can include letters or numbers
    - Can include special characters such as dollar sign, underscore, and pound sign
    - Must limit the length to 30 characters
    - Must not be reserved words

Declaring and Initializing PL/SQL Variables

Syntax:

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

Examples:

```
DECLARE
  emp_hiredate    DATE;
  emp_deptno      NUMBER(2) NOT NULL := 10;
  location        VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

Guidelines for Declaring PL/SQL Variables

- Avoid using column names as identifiers.

```
DECLARE
  employee_id  NUMBER(6);
BEGIN
  SELECT    employee_id
  INTO      employee_id
  FROM      employees
  WHERE     last_name = 'Kochhar';
END;
/
```

- Use the `NOT NULL` constraint when the variable must hold a value.

Base Scalar Data Types

- CHAR [(maximum_length)]
- VARCHAR2 (maximum_length)
- LONG
- NUMBER [(precision, scale)]
- BOOLEAN
- DATE
- TIMESTAMP

Declaring Scalar Variables

- Examples:

```
DECLARE
  emp_job          VARCHAR2(9);
  dept_total_sal   NUMBER(9,2) := 0;
  orderdate        DATE := SYSDATE + 7;
  c_tax_rate       CONSTANT NUMBER(3,2) := 8.25;
  valid            BOOLEAN NOT NULL := TRUE;
  ...
```

Declaring Variables with the %TYPE Attribute

Syntax:

```
identifier      table.column_name%TYPE;
```

- Examples:

```
DATE
TIMESTAMP
```

```
...
  emp_lname       employees.last_name%TYPE;
  balance         NUMBER(7,2);
  min_balance     balance%TYPE := 1000;
...
```

Writing Executable Statements

PL/SQL Block Syntax and Guidelines

- Literals:
  - Character and date literals must be enclosed in single quotation marks.

```
name := 'Henderson';
```

  - Numbers can be simple values or scientific notation.
- Statements can continue over several lines.

Commenting Code

- Prefix single-line comments with two dashes (--).
- Place multiple-line comments between the symbols "/*" and "*/".
- Example:

```
DECLARE
...
annual_sal NUMBER (9,2);
BEGIN    -- Begin the executable section

/* Compute the annual salary based on the
   monthly salary input from the user */
annual_sal := monthly_sal * 12;
END;     -- This is the end of the block
/
```

SQL Functions in PL/SQL: Examples

- Get the length of a string:

```
desc_size INTEGER(5);
prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
desc_size:= LENGTH(prod_description);
```

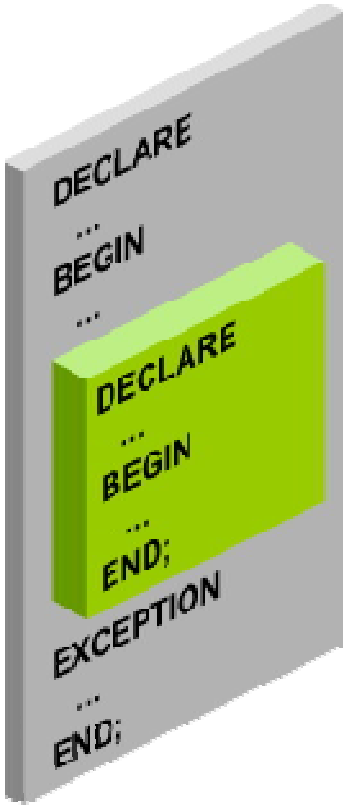
- Convert the employee name to lowercase:

```
emp_name:= LOWER(emp_name);
```

Nested Blocks

PL/SQL blocks can be nested.

- An executable section (`BEGIN ... END`) can contain nested blocks.
- An exception section can contain nested blocks.

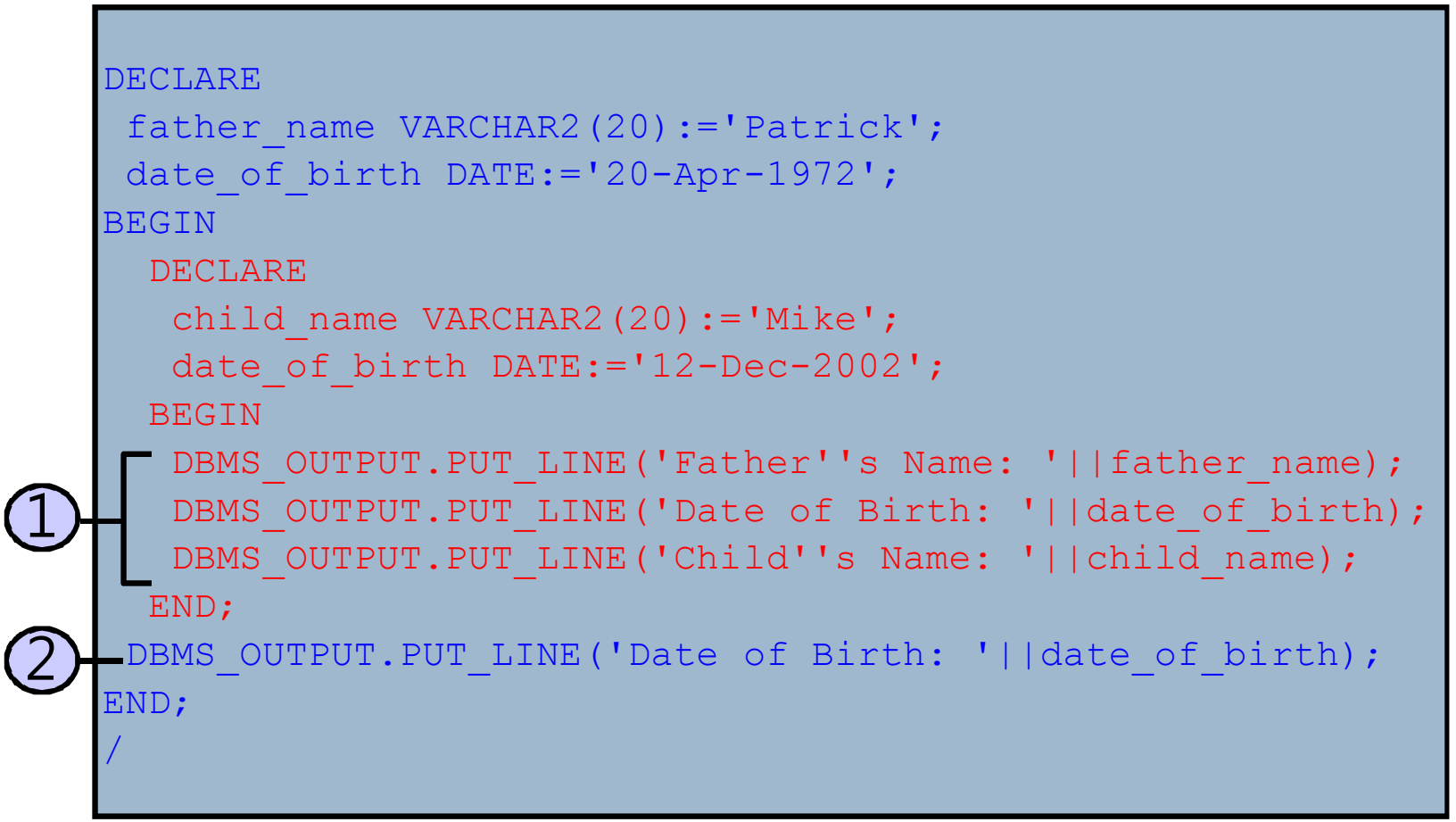


Nested Blocks

Example:

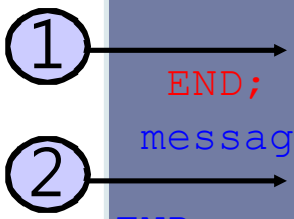
```
DECLARE
 outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
   inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
   DBMS_OUTPUT.PUT_LINE(inner_variable);
   DBMS_OUTPUT.PUT_LINE(outer_variable);
  END;
 DBMS_OUTPUT.PUT_LINE(outer_variable);
END;
/
```

Variable Scope and Visibility



Determining Variable Scope

```
<<outer>>
DECLARE
  sal       NUMBER(7,2) := 60000;
  comm      NUMBER(7,2) := sal * 0.20;
  message   VARCHAR2(255) := ' eligible for commission';
BEGIN
  DECLARE
        sal          NUMBER(7,2) := 50000;
        comm         NUMBER(7,2) := 0;
        total_comp   NUMBER(7,2) := sal + comm;
  BEGIN
        message := 'CLERK not'||message;
        outer.comm := sal * 0.30;
  END;
 message := 'SALESMAN'||message;
END;
/
```



Operators in PL/SQL

- Examples:
  - Increment the counter for a loop.

```
loop_count := loop_count + 1;
```

  - Set the value of a Boolean flag.

```
good_sal := sal BETWEEN 50000 AND 150000;
```

  - Validate whether an employee number contains a value.

```
valid  := (empno IS NOT NULL);
```

Programming Guidelines

- Make code maintenance easier by:
  - Documenting code with comments
  - Developing a case convention for the code
  - Developing naming conventions for identifiers and other objects
  - Enhancing readability by indenting

Indenting Code

- For clarity, indent each level of code.
- Example:

```
BEGIN
   IF x=0 THEN
      y:=1;
   END IF;
END;
/
```

```
DECLARE
   deptno        NUMBER(4);
   location_id   NUMBER(4);
BEGIN
   SELECT   department_id,
            location_id
   INTO     deptno,
            location_id
   FROM     departments
   WHERE    department_name
            = 'Sales';
...
END;
/
```

Practice

- Output variables

```
DECLARE
   customer       VARCHAR2(50) := 'Womansport';
   credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
       customer     NUMBER(7) := 201;
       name   VARCHAR2(25) := 'Unisports';
    BEGIN
       credit_rating :='GOOD';
       …
    END;
  …
END;
/
```

Interacting with the Oracle Server

SELECT Statements in PL/SQL

- Retrieve data from the database with a SELECT statement.
- Syntax:

```
SELECT   select_list
INTO     {variable_name[, variable_name]...
         | record_name}
FROM     table
[WHERE   condition];
```

Retrieving Data in PL/SQL

- Retrieve the `hire_date` and the `salary` for the specified employee.
- Example:

```
DECLARE
 emp_hiredate    employees.hire_date%TYPE;
 emp_salary      employees.salary%TYPE;
BEGIN
  SELECT    hire_date, salary
  INTO      emp_hiredate, emp_salary
  FROM      employees
  WHERE     employee_id = 100;
END;
/
```

Retrieving Data in PL/SQL

- Return the sum of the salaries for all the employees in the specified department.
- Example:

```
SET SERVEROUTPUT ON
DECLARE
   sum_sal  NUMBER(10,2);
   deptno   NUMBER NOT NULL := 60;
BEGIN
   SELECT  SUM(salary)  -- group function
   INTO sum_sal FROM employees
   WHERE  department_id = deptno;
   DBMS_OUTPUT.PUT_LINE ('The sum of salary is '
   || sum_sal);
END;
/
```

Naming Conventions

```
DECLARE
  hire_date       employees.hire_date%TYPE;
  sysdate         hire_date%TYPE;
  employee_id     employees.employee_id%TYPE :=
176;
BEGIN
  SELECT   hire_date, sysdate
  INTO     hire_date, sysdate
  FROM     employees
  WHERE    employee_id = employee_id;
END;
/
```

```
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
```

Inserting Data

- Add new employee information to the `EMPLOYEES` table.
- Example:

```
BEGIN
 INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
   VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
   'RCORES',sysdate, 'AD_ASST', 4000);
END;
/
```

Updating Data

- Increase the salary of all employees who are stock clerks.
- Example:

```
DECLARE
  sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE      employees
  SET         salary = salary + sal_increase
  WHERE       job_id = 'ST_CLERK';
END;
/
```

Deleting Data

- Delete rows that belong to department 10 from the `employees` table.
- Example:

```
DECLARE
  deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   employees
  WHERE  department_id = deptno;
END;
/
```

SQL Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle server.
- There are two types of cursors:
  - Implicit cursors: Created and managed internally by the Oracle server to process SQL statements
  - Explicit cursors: Explicitly declared by the programmer

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 returned at least one row.
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement did not return even one row.
SQL%ROWCOUNT	An integer value that represents 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:

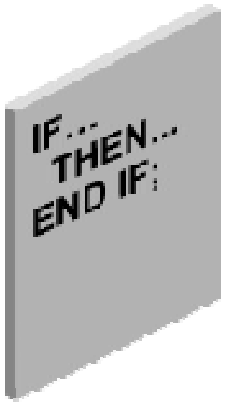
```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
  empno employees.employee_id%TYPE := 176;
BEGIN
  DELETE FROM  employees
  WHERE employee_id = empno;
  :rows_deleted := (SQL%ROWCOUNT ||
                        ' row deleted.');
END;
/
PRINT rows_deleted
```

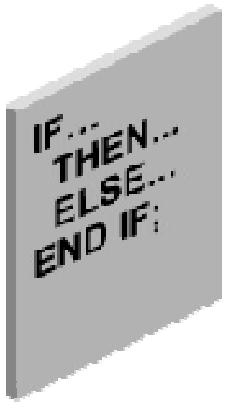
Practice

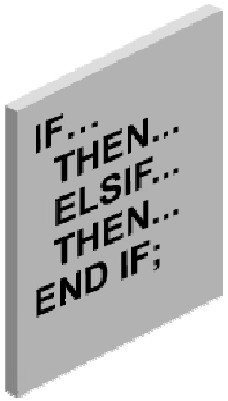
- Create a PL/SQL block that selects the maximum department ID in the departments table and stores it in the max_deptno variable. Display the maximum department ID.

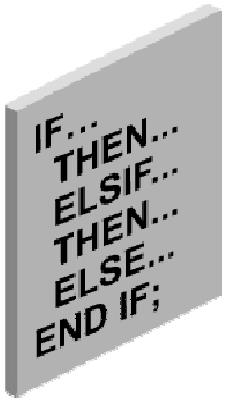
Writing Control Structures

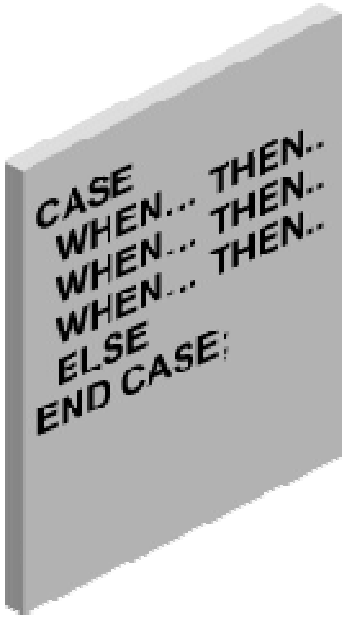
Controlling Flow of Execution

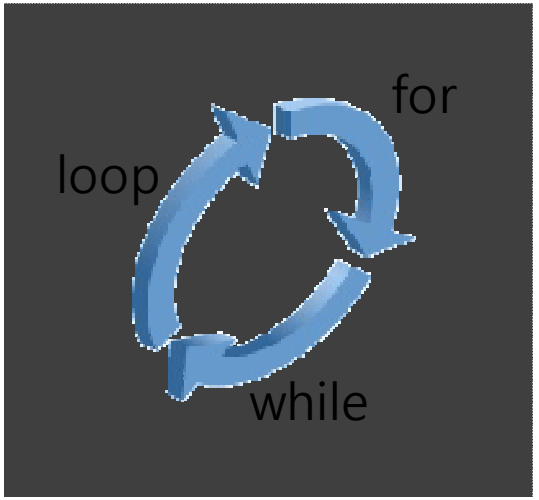












IF Statements

Syntax:

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

Simple IF Statement

```
DECLARE
  myage number:=31;
BEGIN
  IF myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  END IF;
END;
/
```

PL/SQL procedure successfully completed.

IF THEN ELSE Statement

```
SET SERVEROUTPUT ON
DECLARE
myage number:=31;
BEGIN
IF myage < 11
 THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
 ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;
END;
/
```

I am not a child
PL/SQL procedure successfully completed.

IF ELSIF ELSE Clause

```
DECLARE
myage number:=31;
BEGIN
IF myage < 11
 THEN
       DBMS_OUTPUT.PUT_LINE(' I am a child ');
   ELSIF myage < 20
     THEN
       DBMS_OUTPUT.PUT_LINE(' I am young ');
   ELSIF myage < 30
     THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
   ELSIF myage < 40
     THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
 ELSE
    DBMS_OUTPUT.PUT_LINE(' I am always young ');
END IF;
END;
/
```

I am in my thirties
PL/SQL procedure successfully completed.

CASE Expressions

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses expressions. The value returned by these expressions is used to select one of several alternatives.

```
CASE selector
   WHEN expression1 THEN result1
   WHEN expression2 THEN result2
   ...
   WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
/
```

CASE Expressions: Example

```
SET SERVEROUTPUT ON
SET VERIFY OFF
DECLARE
   grade CHAR(1) := UPPER('&grade');
   appraisal VARCHAR2(20);
BEGIN
   appraisal :=
      CASE grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
DBMS_OUTPUT.PUT_LINE ('Grade: '|| grade || '
                      Appraisal ' || appraisal);
END;
/
```

Searched CASE Expressions

```
DECLARE
   grade CHAR(1) := UPPER('&grade');
   appraisal VARCHAR2(20);
BEGIN
    appraisal :=
     CASE
         WHEN grade = 'A' THEN 'Excellent'
         WHEN grade IN ('B','C') THEN 'Good'
         ELSE 'No such grade'
     END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| grade || '
                  Appraisal ' || appraisal);
END;
/
```

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

Boolean Conditions

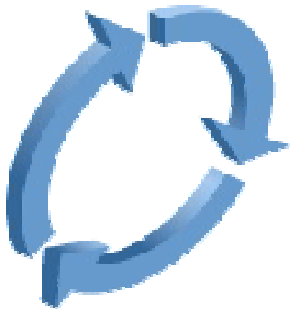
- What is the value of `flag` in each case?

```
flag := reorder_flag AND available_flag;
```

REORDER_FLAG	AVAILABLE_FLAG	FLAG
TRUE	TRUE	?
TRUE	FALSE	?
NULL	TRUE	?
NULL	FALSE	?

Iterative Control: `LOOP` Statements

- Loops repeat a statement or sequence of statements multiple times.
- There are three loop types:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop



Basic Loops

- Syntax:

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

Basic Loops

- Example:

```
DECLARE
  countryid    locations.country_id%TYPE := 'CA';
  loc_id       locations.location_id%TYPE;
  counter       NUMBER(2) := 1;
  new_city     locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO loc_id FROM locations
  WHERE country_id = countryid;
  LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((loc_id + counter), new_city, countryid);
    counter := counter + 1;
    EXIT WHEN counter > 3;
  END LOOP;
END;
/
```

WHILE Loops

- Syntax:

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- Use the WHILE loop to repeat statements while a condition is TRUE.

WHILE Loops

- Example:

```
DECLARE
  countryid   locations.country_id%TYPE := 'CA';
  loc_id      locations.location_id%TYPE;
  new_city    locations.city%TYPE := 'Montreal';
  counter     NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO loc_id FROM locations
  WHERE country_id = countryid;
  WHILE counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((loc_id + counter), new_city, countryid);
    counter := counter + 1;
  END LOOP;
END;
/
```

FOR Loops

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.
- '`lower_bound .. upper_bound`' is required syntax.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

FOR Loops

- Example:

```
DECLARE
  countryid   locations.country_id%TYPE := 'CA';
  loc_id      locations.location_id%TYPE;
  new_city    locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO loc_id
    FROM locations
    WHERE country_id = countryid;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((loc_id + i), new_city, countryid );
  END LOOP;
END;
/
```

Nested Loops and Labels

- Nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the `EXIT` statement that references the label.

Nested Loops and Labels

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    counter := counter+1;
  EXIT WHEN counter>10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT Outer_loop WHEN total_done = 'YES';
      -- Leave both loops
      EXIT WHEN inner_done = 'YES';
      -- Leave inner loop only
      ...
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
/
```

Practice

- Output 0 to 5 with while

Working with Composite Data Types

Creating a PL/SQL Record

- Syntax:

1
```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
```

2
```
identifier    type_name;
```

field_declaration:

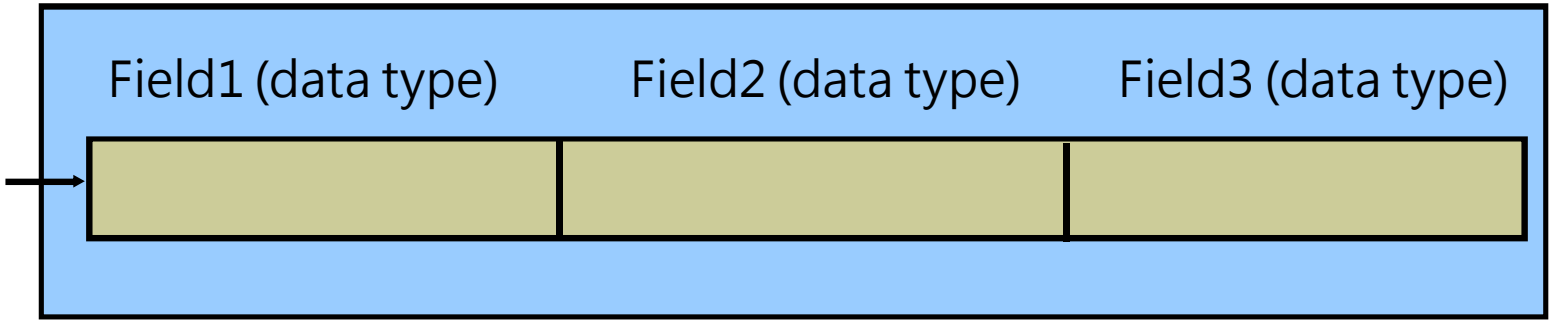
```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

Creating a PL/SQL Record

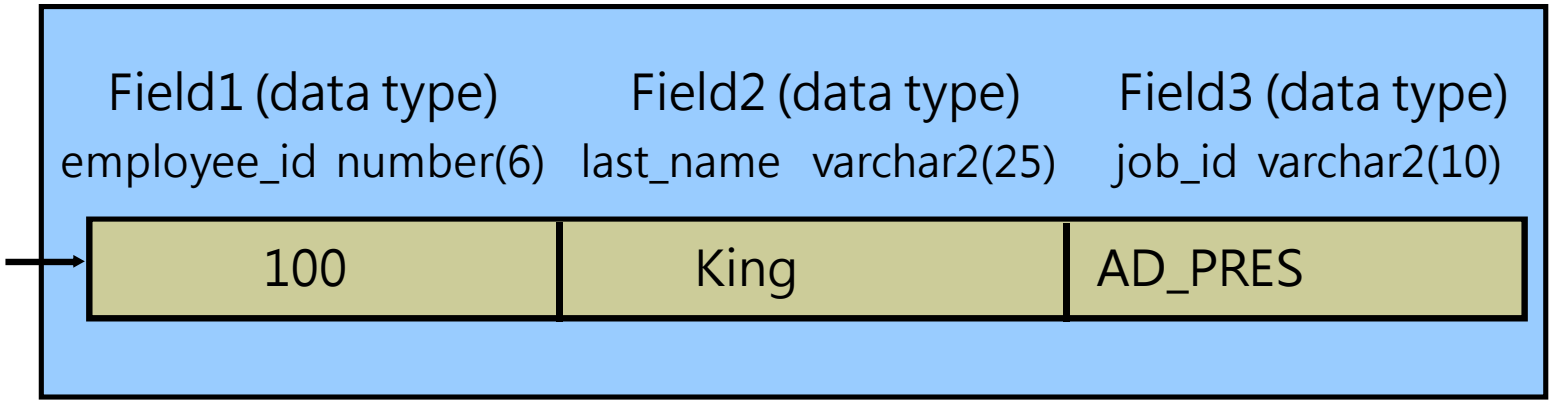
- Declare variables to store the name, job, and salary of a new employee.
- Example:

```
...
  TYPE emp_record_type IS RECORD
    (last_name   VARCHAR2(25),
     job_id      VARCHAR2(10),
     salary      NUMBER(8,2));
  emp_record     emp_record_type;
...
```

PL/SQL Record Structure



Example:



The %ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table or view.
- Fields in the record take their names and data types from the columns of the table or view.

Syntax:

```
DECLARE
   identifier  reference%ROWTYPE;
```

The %ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table or view.
- Fields in the record take their names and data types from the columns of the table or view.

Syntax:

```
DECLARE
   identifier  reference%ROWTYPE;
```

The %ROWTYPE Attribute

```
...
DEFINE employee_number = 124
 DECLARE
  emp_rec   employees%ROWTYPE;
 BEGIN
  SELECT * INTO emp_rec FROM employees
  WHERE  employee_id = &employee_number;
  INSERT INTO retired_emps(empno, ename, job, mgr,
  hiredate, leavedate, sal, comm, deptno)
  VALUES (emp_rec.employee_id, emp_rec.last_name,
  emp_rec.job_id,emp_rec.manager_id,
  emp_rec.hire_date, SYSDATE, emp_rec.salary,
  emp_rec.commission_pct, emp_rec.department_id);
END;
/
```

Inserting a Record Using %ROWTYPE

```
...
DEFINE employee_number = 124
DECLARE
   emp_rec  retired_emps%ROWTYPE;
BEGIN
 SELECT employee_id, last_name, job_id, manager_id,
 hire_date, hire_date, salary, commission_pct,
 department_id INTO emp_rec FROM employees
 WHERE  employee_id = &employee_number;
 INSERT INTO retired_emps VALUES emp_rec;
END;
/
SELECT * FROM retired_emps;
```

Updating a Row in a Table Using a Record

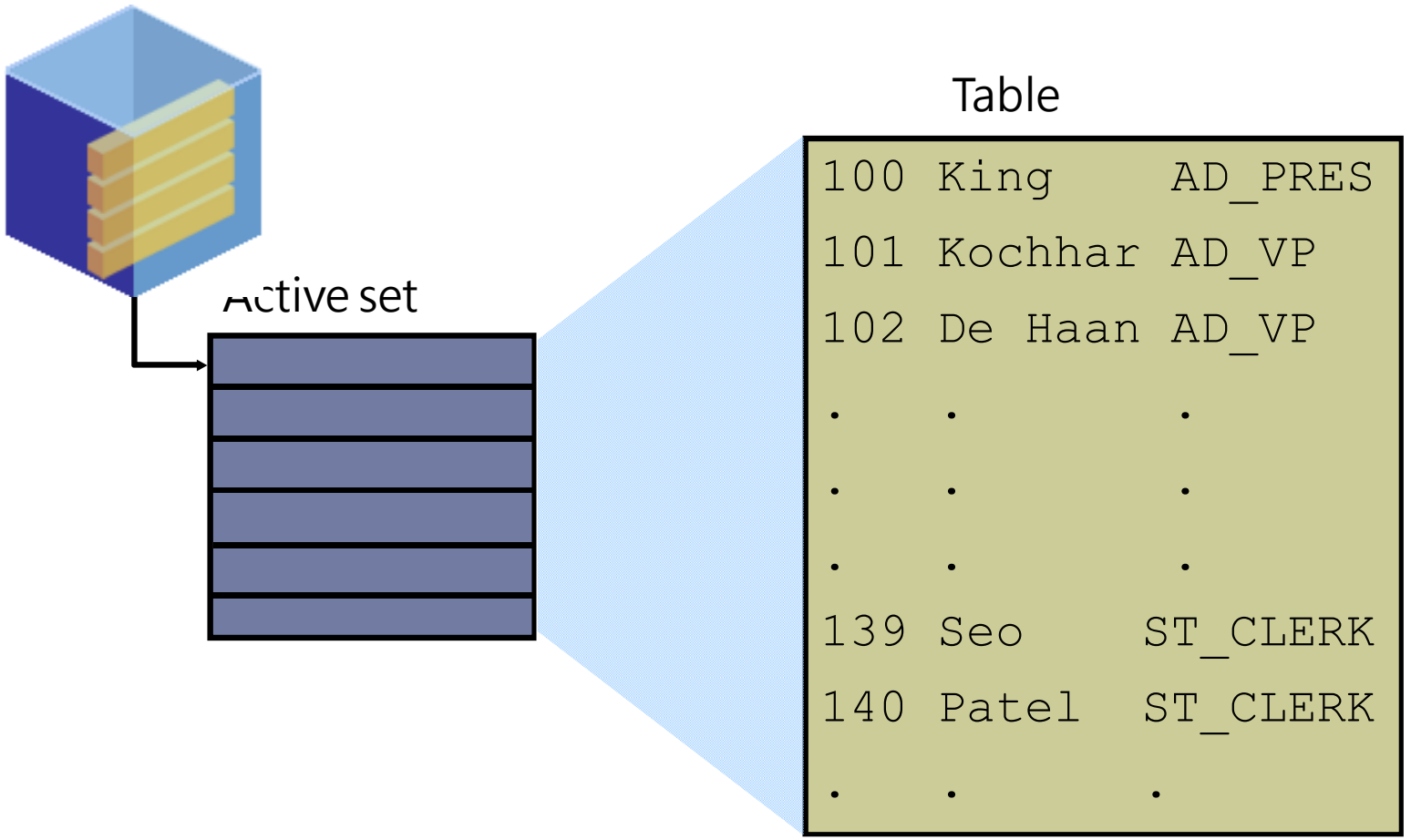
```
SET SERVEROUTPUT ON
SET VERIFY OFF
DEFINE employee_number = 124
DECLARE
   emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO emp_rec FROM retired_emps;
 emp_rec.leavedate:=SYSDATE;
 UPDATE retired_emps SET ROW = emp_rec WHERE
  empno=&employee_number;
END;
/
SELECT * FROM retired_emps;
```

Practice

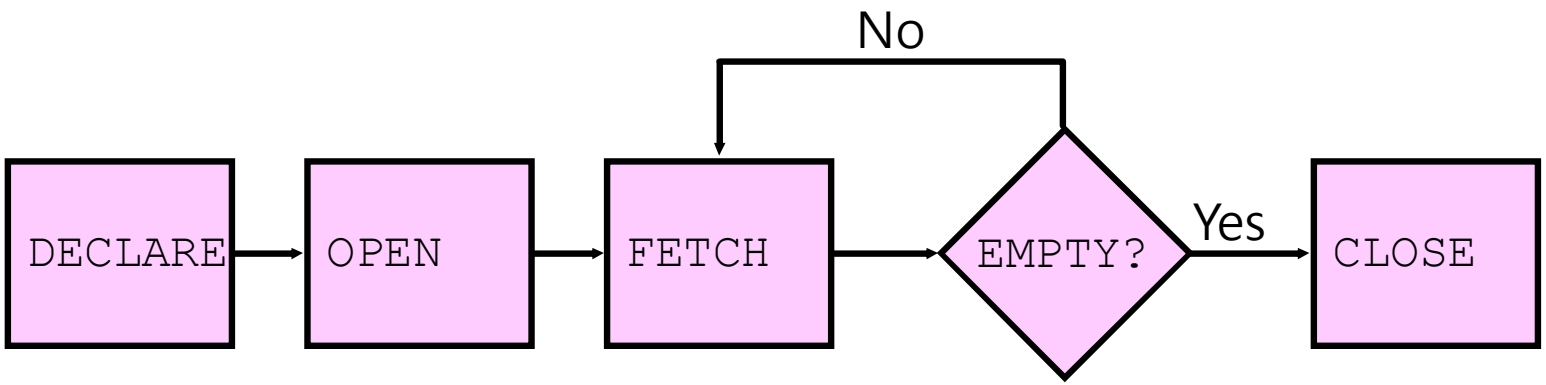
- Select one row (location_id = 1000) from table HR.LOCATIONS with %ROWTYPE and Output CITY value

Using Explicit Cursors

Explicit Cursor Operations



Controlling Explicit Cursors



- Create a named SQL area
- Identify the active set
- Load the current row into variables
- Test for existing rows
- Return to `FETCH` if rows are found
- Release the active set

Declaring the Cursor

- Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

Examples:

```
DECLARE
  CURSOR emp_cursor IS
  SELECT employee_id, last_name FROM employees
  WHERE department_id =30;
```

```
DECLARE
  locid NUMBER:= 1700;
  CURSOR dept_cursor IS
  SELECT * FROM departments
  WHERE location_id = locid;
...
```

Opening the Cursor

```
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
...
BEGIN
  OPEN emp_cursor;
```

Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
  empno employees.employee_id%TYPE;
  lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor INTO empno, lname;
  DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  ...
END;
/
```

Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE  department_id =30;
  empno employees.employee_id%TYPE;
  lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO empno, lname;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  END LOOP;
  ...
END;
/
```

Closing the Cursor

```
...
  LOOP
    FETCH emp_cursor INTO empno, lname;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  END LOOP;
 CLOSE emp_cursor;
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 FOR Loops

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
BEGIN
   FOR emp_record IN emp_cursor
    LOOP
     DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
     ||' ' ||emp_record.last_name);
    END LOOP;
END;
/
```

Explicit Cursor Attributes

- Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far

The %ISOPEN Attribute

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

- Example:

```
IF NOT emp_cursor%ISOPEN THEN
    OPEN emp_cursor;
END IF;
LOOP
  FETCH emp_cursor...
```

Example of %ROWCOUNT and %NOTFOUND

```
SET SERVEROUTPUT ON
DECLARE
  empno  employees.employee_id%TYPE;
  ename  employees.last_name%TYPE;
  CURSOR emp_cursor IS SELECT employee_id,
  last_name FROM employees;
BEGIN
  OPEN emp_cursor;
  LOOP
   FETCH emp_cursor INTO empno, ename;
   EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                      emp_cursor%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(empno)
                        ||'-'|| ename);
  END LOOP;
  CLOSE emp_cursor;
END ;
/
```

Cursor FOR Loops Using Subqueries

- No need to declare the cursor.
- Example:

```
SET SERVEROUTPUT ON
BEGIN
  FOR emp_record IN (SELECT employee_id, last_name
   FROM employees WHERE department_id =30)
  LOOP
   DBMS_OUTPUT.PUT_LINE( emp_record.employee_id ||'
   '||emp_record.last_name);
  END LOOP;
END;
/
```

Cursors with Parameters

- Syntax:

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

  - Pass parameter values to a cursor when the cursor is opened and the query is executed.
  - Open an explicit cursor several times with a different active set each time.

```
OPEN  cursor_name(parameter_value,.....) ;
```

Cursors with Parameters

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR   emp_cursor (deptno NUMBER) IS
   SELECT  employee_id, last_name
   FROM    employees
   WHERE   department_id = deptno;
   dept_id NUMBER;
   lname   VARCHAR2(15);
BEGIN
  OPEN emp_cursor (10);
  ...
  CLOSE emp_cursor;
  OPEN emp_cursor (20);
  ...
```

Practice

- Output all LAST_NAME of EMPLOYEES with cursor

Handling Exceptions

Example

```
SET SERVEROUTPUT ON
DECLARE
  lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO lname FROM employees WHERE
  first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is : '
  ||lname);
END;
/
```

Example

```
SET SERVEROUTPUT ON
DECLARE
  lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO lname FROM employees WHERE
  first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is : '
  ||lname);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE (' Your select statement
  retrieved multiple rows. Consider using a
  cursor.');
END;
/
```

Trapping Exceptions

- Syntax:

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

Guidelines for Trapping Exceptions

- The `EXCEPTION` keyword starts the exception handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.
- `WHEN OTHERS` is the last clause.

Predefined Exceptions

Exception Name	Oracle Server Error Number	Description
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID fails because the character string does not represent a valid ROWID.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Functions for Trapping Exceptions

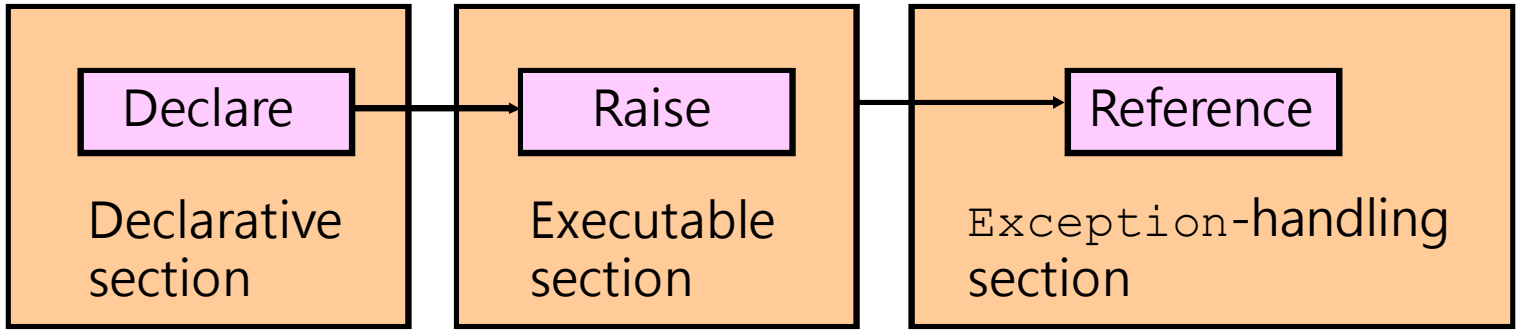
- `SQLCODE`: Returns the numeric value for the error code
- `SQLERRM`: Returns the message associated with the error number

Functions for Trapping Exceptions

- Example:

```
DECLARE
  error_code      NUMBER;
  error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    error_code := SQLCODE ;
    error_message := SQLERRM ;
   INSERT INTO errors (e_user, e_date, error_code,
   error_message) VALUES(USER,SYSDATE,error_code,
   error_message);
END;
/
```

Trapping User-Defined Exceptions



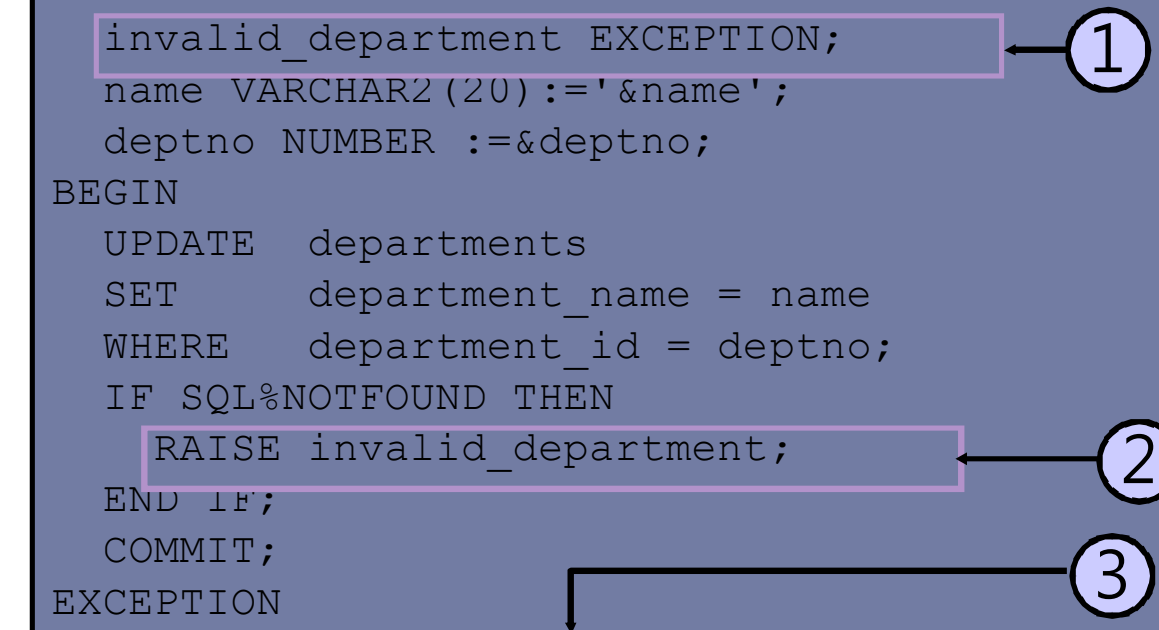
Name the exception.

Explicitly raise the exception by using the `RAISE` statement.

Handle the raised exception.

Trapping User-Defined Exceptions

```
...
ACCEPT deptno PROMPT 'Please enter the department number:'
ACCEPT name   PROMPT 'Please enter the department name:'
DECLARE
  invalid_department EXCEPTION;
  name VARCHAR2(20):='&name';
  deptno NUMBER :=&deptno;
BEGIN
  UPDATE  departments
  SET     department_name = name
  WHERE   department_id = deptno;
  IF SQL%NOTFOUND THEN
    RAISE invalid_department;
  END IF;
  COMMIT;
EXCEPTION
  WHEN invalid_department THEN
    DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
/
```



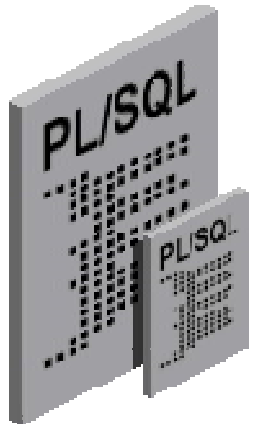
Practice

- Create a ZERO_DIVIDE exception and handle it.

Creating Stored Procedures and Functions

Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional declarative section (without `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



Procedure: Syntax

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

Procedure: Example

```
...
CREATE TABLE dept AS SELECT * FROM departments;
CREATE PROCEDURE add_dept IS
 dept_id dept.department_id%TYPE;
 dept_name dept.department_name%TYPE;
BEGIN
 dept_id:=280;
 dept_name:='ST-Curriculum';
 INSERT INTO dept(department_id,department_name)
 VALUES(dept_id,dept_name);
 DBMS_OUTPUT.PUT_LINE(' Inserted '||
  SQL%ROWCOUNT ||' row ');
END;
/
```

Invoking the Procedure

```
BEGIN
 add_dept;
END;
/
SELECT department_id, department_name FROM
dept WHERE department_id=280;
```

Inserted 1 row
PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME
280	ST-Curriculum

Function: Syntax

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

Function: Example

```
CREATE FUNCTION check_sal RETURN Boolean IS
 dept_id employees.department_id%TYPE;
 empno   employees.employee_id%TYPE;
 sal     employees.salary%TYPE;
 avg_sal employees.salary%TYPE;
BEGIN
 empno:=205;
 SELECT salary,department_id INTO sal,dept_id
 FROM employees WHERE employee_id= empno;
 SELECT avg(salary) INTO avg_sal FROM employees
 WHERE department_id=dept_id;
 IF sal > avg_sal THEN
  RETURN TRUE;
 ELSE
  RETURN FALSE;
 END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RETURN NULL;
END;
/
```

Invoking the Function

```
SET SERVEROUTPUT ON
BEGIN
 IF (check_sal IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
END;
/
```

```
Salary > average
PL/SQL procedure successfully completed.
```

Passing Parameter to the Function

```
DROP FUNCTION check_sal;
/
CREATE FUNCTION check_sal(empno employees.employee_id%TYPE)
RETURN Boolean IS
 dept_id employees.department_id%TYPE;
 sal     employees.salary%TYPE;
 avg_sal employees.salary%TYPE;
BEGIN
 SELECT salary,department_id INTO sal,dept_id
 FROM employees WHERE employee_id=empno;
 SELECT avg(salary) INTO avg_sal FROM employees
 WHERE department_id=dept_id;
 IF sal > avg_sal THEN
  RETURN TRUE;
 ELSE
  RETURN FALSE;
 END IF;
EXCEPTION ...
...
```

Invoking the Function with a Parameter

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 205');
 IF (check_sal(205) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(205)) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 70');
 IF (check_sal(70) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(70)) THEN
 ...
 END IF;
END;
/
```

Procedures VS Function

- Function return value, procedure can not
- Function can call from SQL statement, procedure can not
- Functions are considered expressions, procedure are not

Practice

- Create a function check_sal_gt10000
  - Pass a employees.employee_id into function
  - To check the employee whether her/his salary is great then 10000 on table EMPLOYEES
- Invoke the function check_sal_gt10000

USER DEFINED TYPE

Object types

- They let you model real-world objects, separate interfaces and implementation details, and store object-oriented data persistently in the database.

Object type example

```
CREATE TYPE address_typ AS OBJECT (
        street VARCHAR2(30),
        city VARCHAR2(20),
        state CHAR(2),
        postal_code VARCHAR2(6) );
```

Object type example

```
CREATE TYPE employee_typ AS OBJECT (
        employee_id NUMBER(6),
        first_name VARCHAR2(20),
        last_name VARCHAR2(25),
        email VARCHAR2(25),
        phone_number VARCHAR2(20),
        hire_date DATE,
        job_id VARCHAR2(10),
        salary NUMBER(8,2),
        commission_pct NUMBER(2,2),
        manager_id NUMBER(6),
        department_id NUMBER(4),
        address address_typ,
        MAP MEMBER FUNCTION get_idno RETURN NUMBER,
        MEMBER PROCEDURE display_address
                ( SELF IN OUT NOCOPY employee_typ ) );
```

Object type example

```
CREATE TYPE BODY employee_typ AS
        MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
        BEGIN
                RETURN employee_id;
        END;

        MEMBER PROCEDURE display_address
                ( SELF IN OUT NOCOPY employee_typ ) IS
        BEGIN
                DBMS_OUTPUT.PUT_LINE(first_name || ' ' || last_name);
                DBMS_OUTPUT.PUT_LINE(address.street);
                DBMS_OUTPUT.PUT_LINE(address.city || ', ' || address.state
                        || ' ' || address.postal_code);
        END;

END;
```

Declaring Object Types

```
DECLARE
        emp employee_typ; -- emp is atomically null
BEGIN
        -- call the constructor for employee_typ
        emp := employee_typ(315, 'Francis', 'Logan', 'FLOGAN',
                '555.777.2222', '01-MAY-04', 'SA_MAN', 11000, .15, 101, 110,
                address_typ('376 Mission', 'San Francisco', 'CA', '94222'));
         -- display details
        DBMS_OUTPUT.PUT_LINE(emp.first_name || ' ' || emp.last_name);
        emp.display_address(); -- call object method to display details
END;
```

Inserting Rows in an Object Table

```
DECLARE
        emp employee_typ;
BEGIN
        INSERT INTO employee_tab VALUES (
                employee_typ(310, 'Evers', 'Boston', 'EBOSTON',
                '555.111.2222', '01-AUG-04', 'SA_REP', 9000, .15, 101, 110,
                address_typ('123 Main', 'San Francisco', 'CA', '94111')) );
        INSERT INTO employee_tab VALUES (
                employee_typ(320, 'Martha', 'Dunn', 'MDUNN',
                '555.111.3333', '30-SEP-04', 'AC_MGR', 12500, 0, 101, 110,
                address_typ('123 Broadway', 'Redwood City', 'CA', '94065')) );
END;
```

Accessing Object Methods

```
DECLARE
        emp employee_typ;
BEGIN
        SELECT VALUE(e) INTO emp FROM employee_tab e
                WHERE e.employee_id = 310;
        emp.display_address();
END;
```

*Note the use of the VALUE function which returns the value of an object.

Updating and Deleting Objects

```
DECLARE
        emp employee_typ;
BEGIN
        INSERT INTO employee_tab VALUES (
                employee_typ(370, 'Robert', 'Myers', 'RMYERS',
                '555.111.2277', '07-NOV-04', 'SA_REP', 8800, .12, 101, 110,
                address_typ('540 Fillmore', 'San Francisco', 'CA', '94011')) );
        UPDATE employee_tab e SET e.address.street = '1040 California'
                WHERE e.employee_id = 370;
        DELETE FROM employee_tab e WHERE e.employee_id = 310;
END;
```

DBMS_LOB

DBMS_LOB

- DBMS_LOB can read and modify BLOBs, CLOBs, and NCLOBs; it provides read-only operations for BFILEs. The bulk of the LOB operations are provided by this package.
- The value of offset, amount, newlen, nth must not exceed the value lobmaxsize (4GB-1) in any DBMS_LOB subprogram.

DBMS_LOB

- length, offset, and amount parameters for subprograms operating on **BLOBs** and **BFILEs** must be specified in terms of **bytes**.
- length, offset, and amount parameters for subprograms operating on **CLOBs** must be specified in terms of **characters**.

Datatypes Used by DBMS_LOB

Type	Description
BLOB	Source or destination binary LOB.
RAW	Source or destination RAW buffer (used with BLOB).
CLOB	Source or destination character LOB (including NCLOB).
VARCHAR2	Source or destination character buffer (used with CLOB and NCLOB).
INTEGER	Specifies the size of a buffer or LOB, the offset into a LOB, or the amount to access.
BFILE	Large, binary object stored outside the database.

Summary of DBMS_LOB Subprograms

Subprogram	Description
OPEN	Opens a LOB (internal, external, or temporary) in the indicated mode
CLOSE	Closes a previously opened internal or external LOB
ISOPEN	Checks to see if the LOB was already opened using the input locator
GETLENGTH	Gets the length of the LOB value
READ	Reads data from the LOB starting at the specified offset
WRITE	Writes data to the LOB from a specified offset

Summary of DBMS_LOB Subprograms

Subprogram	Description
FILEOPEN	Opens a file
FILECLOSE	Closes the file
FILEISOPEN	Checks if the file was opened using the input BFILE locators
FILEGETNAME	Gets the directory object name and file name
LOADBLOBFROMFILE	Loads BFILE data into an internal BLOB
LOADCLOBFROMFILE	Loads BFILE data into an internal CLOB
LOADFROMFILE	Loads BFILE data into an internal LOB

TRIGGER

TRIGGER

- You can write triggers that fire whenever one of the following operations occurs:
  1. DML statements (INSERT, UPDATE, DELETE) on a particular table or view, issued by any user
  2. DDL statements (CREATE or ALTER primarily) issued either by a particular schema/user or by any schema/user in the database
  3. Database events, such as logon/logoff, errors, or startup/shutdown, also issued either by a particular schema/user or by any schema/user in the database

Designing Triggers

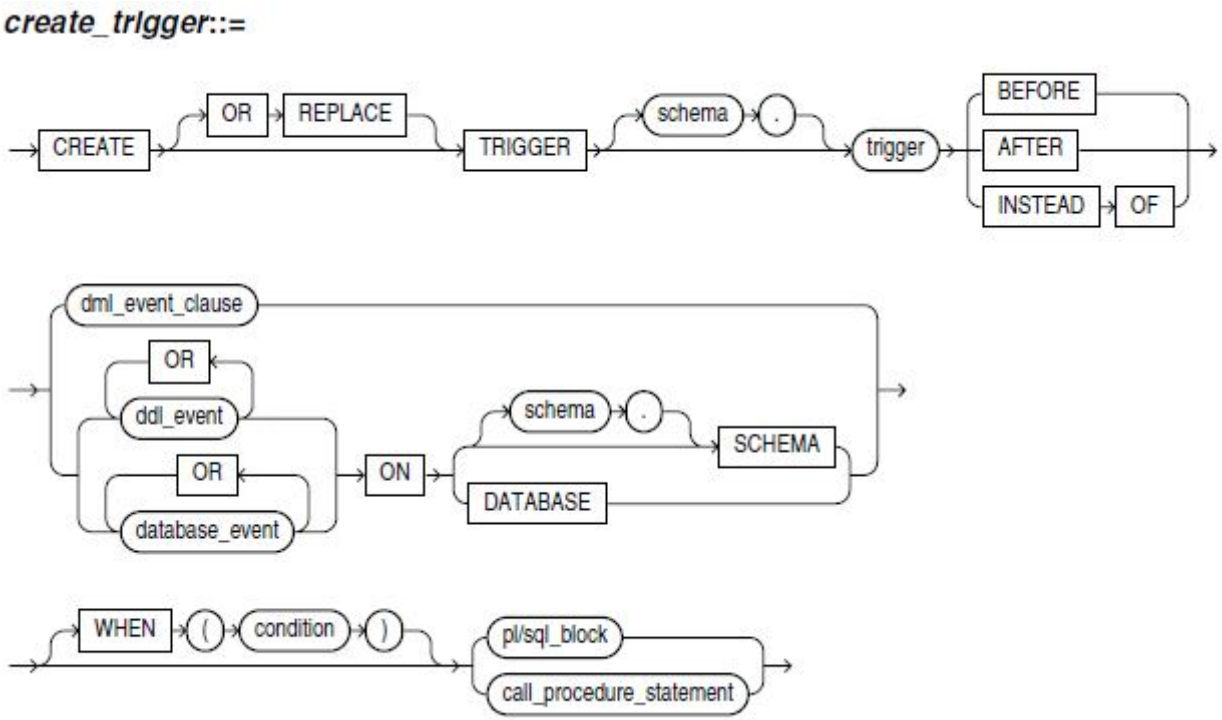
- Use the following guidelines when designing your triggers:
  - Use triggers to guarantee that when a specific operation is performed, related actions are performed.
  - Do not define triggers that duplicate features already built into Oracle Database.

    For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints.
  - Limit the size of triggers. If the logic for your trigger requires much more than 60 lines of PL/SQL code, it is better to include most of the code in a stored procedure and call the procedure from the trigger.

Designing Triggers

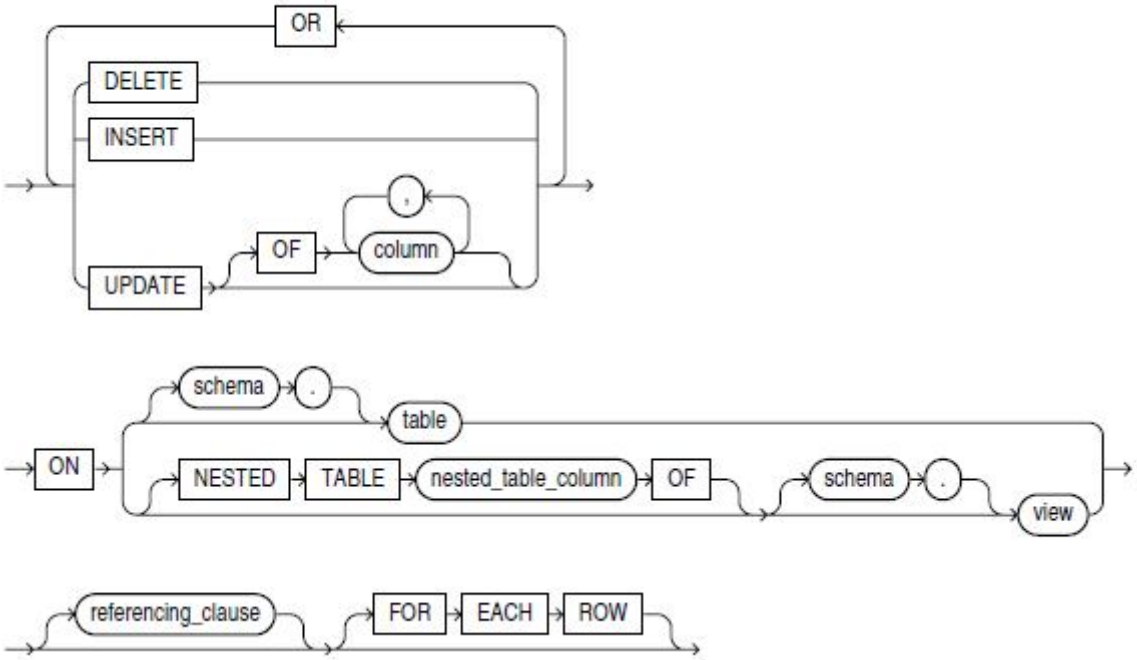
- Do not create recursive triggers. For example, creating an AFTER UPDATE statement trigger on the Emp_tab table that itself issues an UPDATE statement on Emp_tab, causes the trigger to fire recursively until it has run out of memory.
- Use triggers on DATABASE judiciously. They are executed for every user every time the event occurs on which the trigger is created.

Create trigger syntax



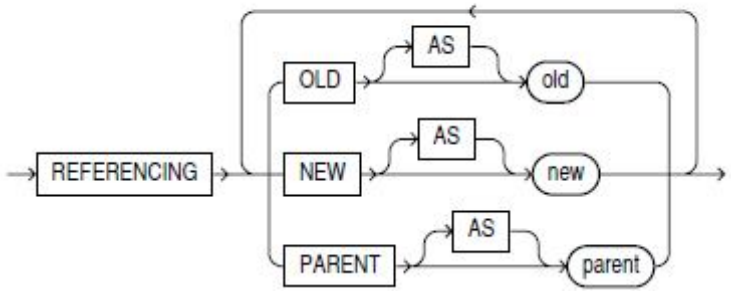
Create trigger syntax

DML_event_clause::=



Create trigger syntax

referencing_clause::=



DML trigger example

```
CREATE TRIGGER hr.salary_check
        BEFORE INSERT OR UPDATE OF salary, job_id ON hr.employees
        FOR EACH ROW
        WHEN (new.job_id <> 'AD_VP')
        pl/sql_block
```

DDL trigger example

```
CREATE TRIGGER audit_db_object AFTER CREATE
        ON SCHEMA
        pl/sql_block
```

Schema event trigger example

```
CREATE OR REPLACE TRIGGER drop_trigger
        BEFORE DROP ON hr.SCHEMA
        BEGIN
                RAISE_APPLICATION_ERROR (
                        num => -20000,
                        msg => 'Cannot drop object');
        END;
```

Database event trigger example

```
CREATE TRIGGER log_errors AFTER SERVERERROR ON DATABASE
        BEGIN
                IF (IS_SERVERERROR (1017)) THEN
                        <special processing of logon error>
                ELSE
                        <log error number>
        END
```

DBMS_SCHEDULER

Rights

- CREATE JOB

Simple Sample

```
begin
  dbms_scheduler.create_job(
       job_name => 'DEMO_JOB_SCHEDULE'
      ,job_type => 'PLSQL_BLOCK'
      ,job_action => 'begin package.procedure(''param_value''); end; '
      ,start_date => '05/25/2009 15:40 PM'
      ,repeat_interval => 'FREQ=DAILY'
      ,enabled => TRUE
      ,comments => 'Demo for job schedule.');
end;
/
```

Program

- The program component represents program-code that can be executed. This program code can have parameters.
- program_type can have one of the following values: 'PLSQL_BLOCK', 'STORED_PROCEDURE', 'EXECUTABLE ‘

Program - Example

```
begin
   dbms_scheduler.create_program (
       program_name => 'DEMO_JOB_SCHEDULE'
      ,program_type => 'STORED_PROCEDURE'
      ,program_action => 'package.procedure'
      ,number_of_arguments => 1
      ,enabled => FALSE
      ,comments => 'Demo for job schedule.');

   dbms_scheduler.define_program_argument (
       program_name => 'DEMO_JOB_SCHEDULE'
      ,argument_position => 1
      ,argument_name => 'kol1'
      ,argument_type => 'VARCHAR2'
      ,default_value => 'default'
    );
    dbms_scheduler.enable(name => 'DEMO_JOB_SCHEDULE');
end;
/
```

Schedule

- A schedule defines the frequence and date/time specifics of the start-time for the job.
- Calendar expresions can have one of these values: 'Yearly', 'Monthly', 'Weekly', 'Daily', 'Hourly', 'Minutely', 'Secondely'

Schedule - example

```
begin
  dbms_scheduler.create_schedule(
      schedule_name => 'DEMO_SCHEDULE'
    , start_date =>  '01/01/2006 22:00:00'
    , repeat_interval => 'FREQ=WEEKLY'
    , comments => 'Weekly at 22:00');
END;
/

To drop the schedule:
begin
  dbms_scheduler.drop_schedule(
     schedule_name => 'DEMO_SCHEDULE'
     , force => TRUE );
end;
/
```

Job

- A job defines when a specific task will be started. This can be done by assigning a program to one or more schedules (or to a specific date/time).

Job - example

```
begin
  dbms_scheduler.create_job(
      job_name => 'DEMO_JOB1'
    , program_name =>'DEMO_JOB_SCHEDULE'
    , schedule_name =>'DEMO_SCHEDULE'
    , enabled => FALSE
    , comments => 'Run demo program every week at 22:00');

  dbms_scheduler.set_job_argument_value(
      job_name => 'DEMO_JOB1'
    , argument_position => 1
    , argument_value => 'param1');

  dbms_scheduler.enable('DEMO_JOB1');

  commit;
end;
/
```

Job – shell script

```
begin
  dbms_scheduler.create_job
  (
    job_name      => 'RUN_SHELL1',
    schedule_name => 'DEMO_SCHEDULE',
    job_type      => 'EXECUTABLE',
    job_action    => '/home/test/run_script.sh',
    enabled       => true,
    comments      => 'Run shell-script'
  );
end;
/
```

Monitoring job-scheduling

- Show details on job run

```
select log_date
,      job_name
,      status
,      req_start_date
,      actual_start_date
,      run_duration
from   dba_scheduler_job_run_details
```

Monitoring job-scheduling

- Show running jobs

```
select job_name
,      session_id
,      running_instance
,      elapsed_time
,      cpu_used
from dba_scheduler_running_jobs;
```

- Show job history

```
select log_date
,      job_name
,      status
from dba_scheduler_job_log;
```

Monitoring job-scheduling

- Show all schedules

```
select schedule_name, schedule_type, start_date, repeat_interval
from dba_scheduler_schedules;
```

- Show all jobs and their attributes

```
select *
from dba_scheduler_jobs
```

- Show all program-objects and their attributes

```
select *
from dba_scheduler_programs;
```

- show all program-arguments

```
select *
from   dba_scheduler_program_args;
```

Lock

Lock

- Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource—either user objects such as tables and rows or system objects not visible to users, such as shared data structures in memory and data dictionary rows.

Lock

- Oracle automatically uses different types of locks to control concurrent access to data and to prevent destructive interaction between users.
- Oracle automatically locks a resource on behalf of a transaction to prevent other transactions from doing something also requiring exclusive access to the same resource.
- The lock is released automatically when some event occurs so that the transaction no longer requires the resource.

Modes of Locking

- Exclusive lock mode
- Share lock mode

Types of Locks

- DML locks (data locks)
  - DML locks protect data. For example, table locks lock entire tables, row locks lock selected rows.
- DDL locks (dictionary locks)
  - DDL locks protect the structure of schema objects—for example, the definitions of tables and views.
- Internal locks and latches
  - Internal locks and latches protect internal database structures such as datafiles. Internal locks and latches are entirely automatic.

DML Locks

- Row Locks (TX)
  - Row-level locks are primarily used to prevent two transactions from modifying the same row. When a transaction needs to modify a row, a row lock is acquired.
  - Readers of data do not wait for writers of the same data rows.
  - Writers of data do not wait for readers of the same data rows unless SELECT ... FOR UPDATE is used, which specifically requests a lock for the reader.
  - Writers only wait for other writers if they attempt to update the same rows at the same time.
  - If a transaction obtains a row lock for a row, the transaction also acquires a table lock for the corresponding table. The table lock prevents conflicting DDL operations that would override data changes in a current transaction.

DML Locks

- Table Locks (TM)
  - Table-level locks are primarily used to do concurrency control with concurrent DDL operations, such as preventing a table from being dropped in the middle of a DML operation.
  - A transaction acquires a table lock when a table is modified in the following DML statements: INSERT, UPDATE, DELETE, SELECT with the FOR UPDATE clause, and LOCK TABLE.

DML Locks - Table Locks

- Row Share Table Locks (RS)
  - subshare table lock, SS

DML Locks - Table Locks

- Row Exclusive Table Locks (RX)
  - subexclusive table lock, SX

DML Locks - Table Locks

- Share Table Locks (S)

DML Locks - Table Locks

- Share Row Exclusive Table Locks (SRX)
  - share-subexclusive table lock, SSX

DML Locks - Table Locks

- Exclusive Table Locks (X)

DML Locks – Table Locks summary

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X
SELECT...FROM *table*...	none	Y	Y	Y	Y	Y
INSERT INTO *table* ...	RX	Y	Y	N	N	N
UPDATE *table* ...	RX	Y*	Y*	N	N	N
DELETE FROM *table* ...	RX	Y*	Y*	N	N	N
SELECT ... FROM *table* FOR UPDATE OF ...	RS	Y*	Y*	Y*	Y*	N
LOCK TABLE *table* IN ROW SHARE MODE	RS	Y	Y	Y	Y	N
LOCK TABLE *table* IN ROW EXCLUSIVE MODE	RX	Y	Y	N	N	N
LOCK TABLE *table* IN SHARE MODE	S	Y	N	Y	N	N
LOCK TABLE *table* IN SHARE ROW EXCLUSIVE MODE	SRX	Y	N	N	N	N
LOCK TABLE *table* IN EXCLUSIVE MODE	X	N	N	N	N	N

DDL Locks

- A data dictionary lock (DDL) protects the definition of a schema object while that object is acted upon or referred to by an ongoing DDL operation.
- The DDL locks prevent objects referenced in the procedure from being altered or dropped before the procedure compilation is complete.
- DDL locks fall into three categories: exclusive DDL locks, share DDL locks, and breakable parse locks.

DDL Locks - Exclusive DDL Locks

- DDL operations require exclusive DDL locks for a resource to prevent destructive interference with other DDL operations that might modify or reference the same schema object.
- During the acquisition of an exclusive DDL lock, if another DDL lock is already held on the schema object by another operation, the acquisition waits until the older DDL lock is released and then proceeds.

DDL Locks - Share DDL Locks

- Some DDL operations require share DDL locks for a resource to prevent destructive interference with conflicting DDL operations, but allow data concurrency for similar DDL operations.
- For example, when a CREATE PROCEDURE statement is run, the containing transaction acquires share DDL locks for all referenced tables. No transaction can alter or drop a referenced table.

DDL Locks - Breakable Parse Locks

- A SQL statement (or PL/SQL program unit) in the shared pool holds a parse lock for each schema object it references.
- A parse lock is acquired during the parse phase of SQL statement execution and held as long as the shared SQL area for that statement remains in the shared pool.
- Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped.

Latches and Internal Locks

- Latches and internal locks protect internal database and memory structures.

Latches

- Latches are simple, low-level serialization mechanisms to protect shared data structures in the system global area (SGA).
- A server or background process acquires a latch for a very short time while manipulating or looking at one of these structures.

Internal Locks

- Dictionary Cache Locks
- File and Log Management Locks
- Tablespace and Rollback Segment Locks

Internal Locks - Dictionary Cache Locks

- Dictionary cache locks can be shared or exclusive.
- Shared locks are released when the parse is complete.
- Exclusive locks are released when the DDL operation is complete.

Internal Locks - File and Log Management Locks

- These locks protect various files.
- protects the control file so that only one process at a time can change it.
- Another lock coordinates the use and archiving of the redo log files.
- Datafiles are locked to ensure that multiple instances mount a database in shared mode or that one instance mounts it in exclusive mode.

Internal Locks - Tablespace and Rollback Segment Locks

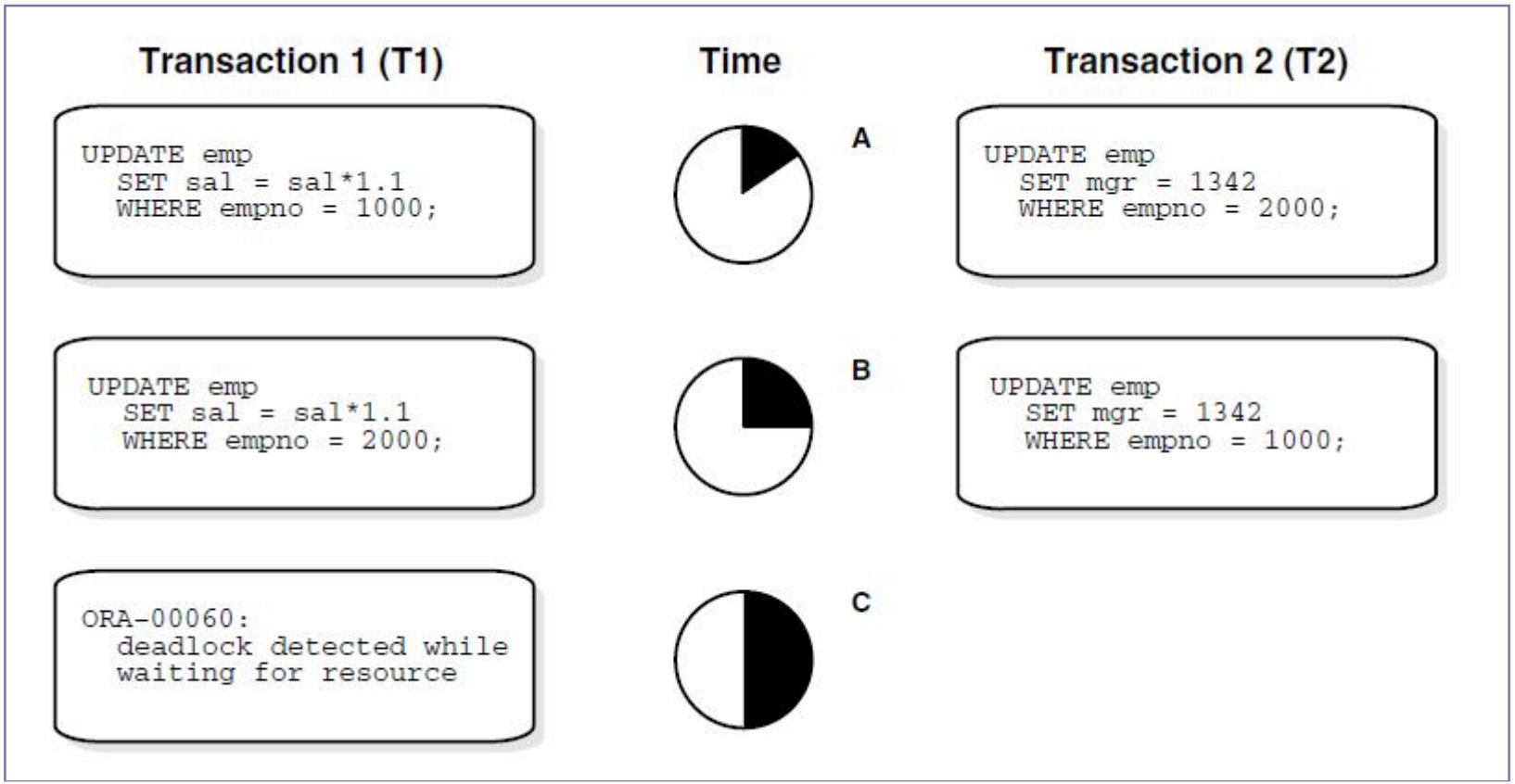
- These locks protect tablespaces and rollback segments.
- all instances accessing a database must agree on whether a tablespace is online or offline.
- Rollback segments are locked so that only one instance can write to a segment.

Explicit (Manual) Data Locking

- At the transaction level, transactions that include the following SQL statements override Oracle's default locking:
  - The SET TRANSACTION ISOLATION LEVEL statement
  - The LOCK TABLE statement (which locks either a table or, when used with views, the underlying base tables)
  - The SELECT ... FOR UPDATE statement

Dead Lock

- A deadlock can occur when two or more users are waiting for data locked by each other.



Dead Lock

- Oracle automatically detects deadlock situations and resolves them by rolling back one of the statements involved in the deadlock, thereby releasing one set of the conflicting row locks.