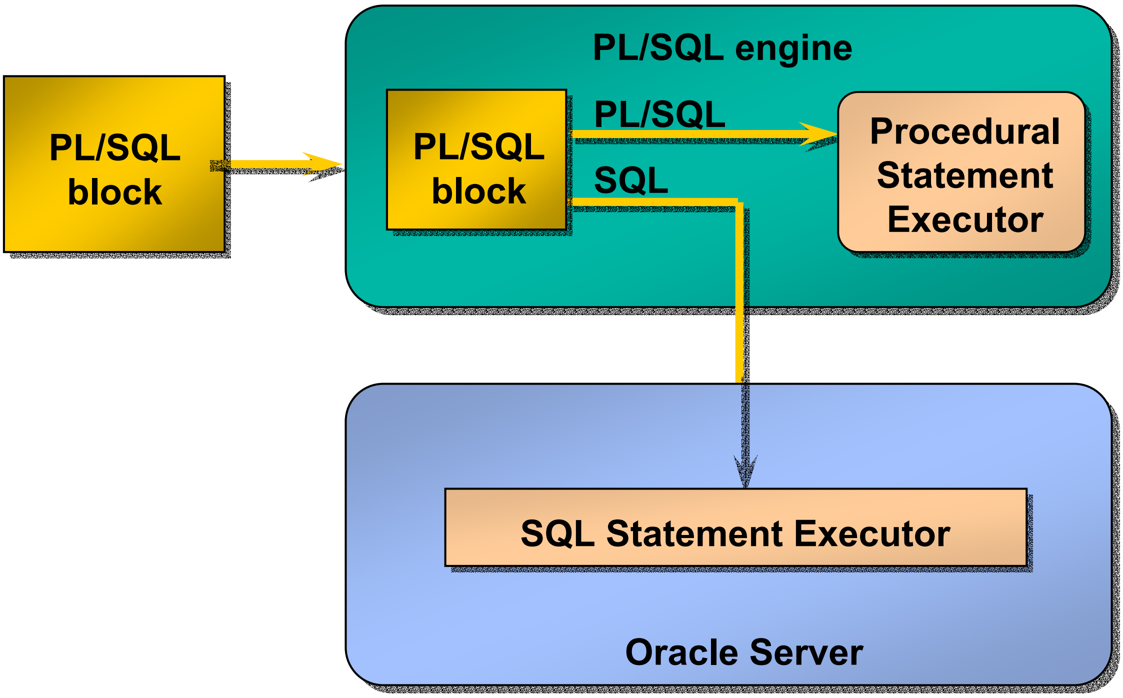
Overview of PL/SQL

About PL/SQL

- PL/SQL is an extension to SQL with design features of programming languages.
- Data manipulation and query statements of SQL are included within procedural units of code.

PL/SQL Environment



PL/SQL block

PL/SQL engine

PL/SQL block

PL/SQL

SQL

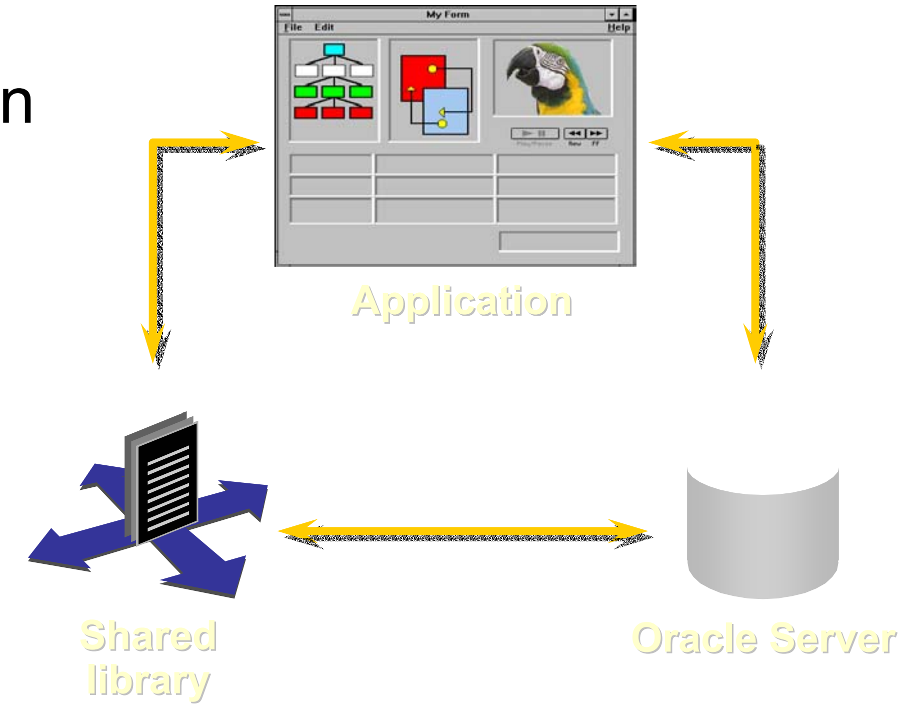
Procedural Statement Executor

SQL Statement Executor

Oracle Server

Benefits of PL/SQL

- Integration



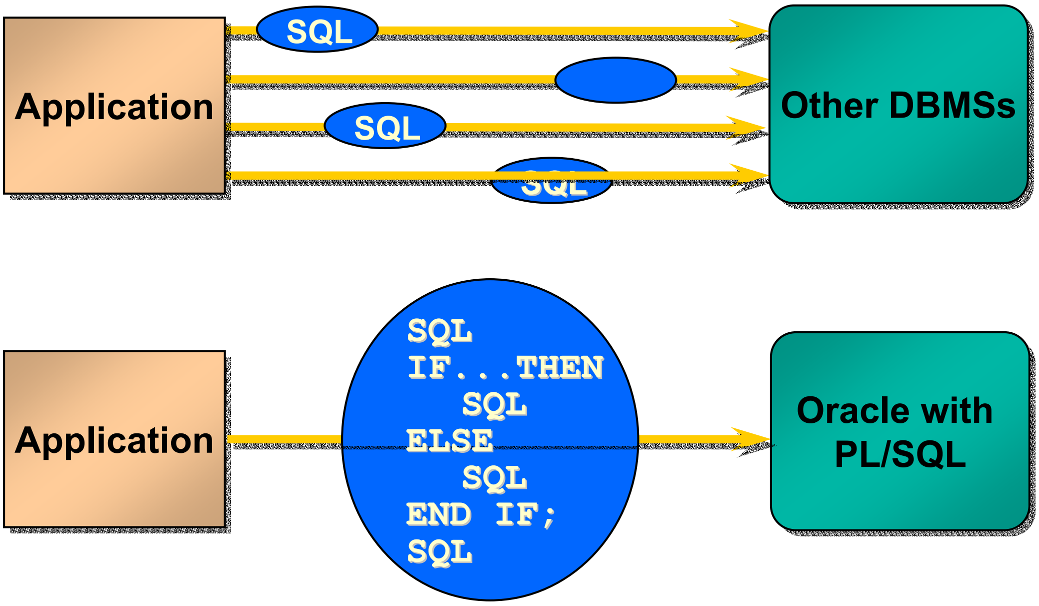
Application

Shared library

Oracle Server

Benefits of PL/SQL

- Improve performance



Application

SQL

SQL

SQL

Other DBMSs

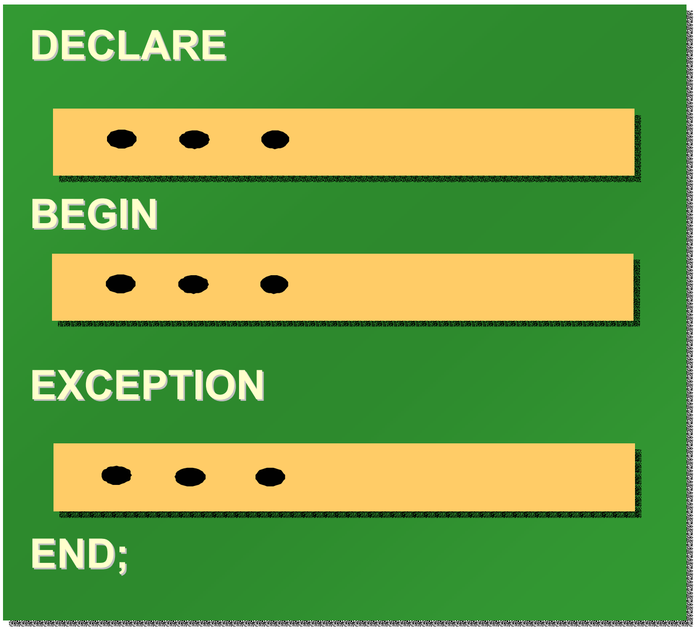
Application

```
SQL
IF...THEN
    SQL
ELSE
    SQL
END IF;
SQL
```

Oracle with PL/SQL

Benefits of PL/SQL

- Modularize program development



DECLARE

• • •

BEGIN

• • •

EXCEPTION

• • •

END;

Benefits of PL/SQL

- It is portable.
- You can declare identifiers.
- You can program with procedural language control structures.
- It can handle errors.

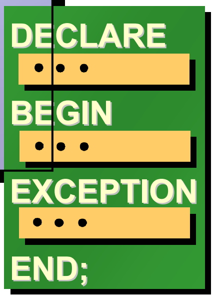
Declaring Variables

PL/SQL Block Structure

- DECLARE – Optional

  Variables, cursors, user-defined exceptions
- BEGIN – Mandatory
  - SQL statements
  - PL/SQL statements
- EXCEPTION – Optional

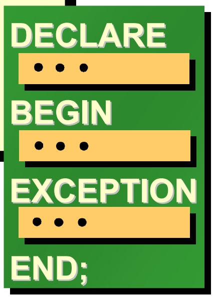
  Actions to perform when errors occur
- END; – Mandatory



DECLARE
• • •
BEGIN
• • •
EXCEPTION
• • •
END;

PL/SQL Block Structure

```
DECLARE
  v_variable  VARCHAR2(5);
BEGIN
  SELECT      column_name
    INTO      v_variable
    FROM      table_name;
EXCEPTION
  WHEN exception_name THEN
  ...
END;
```



DECLARE
• • •
BEGIN
• • •
EXCEPTION
• • •
END;

Block Types

- Anonymous

```
[DECLARE]


BEGIN
   --statements

[EXCEPTION]

END;
```

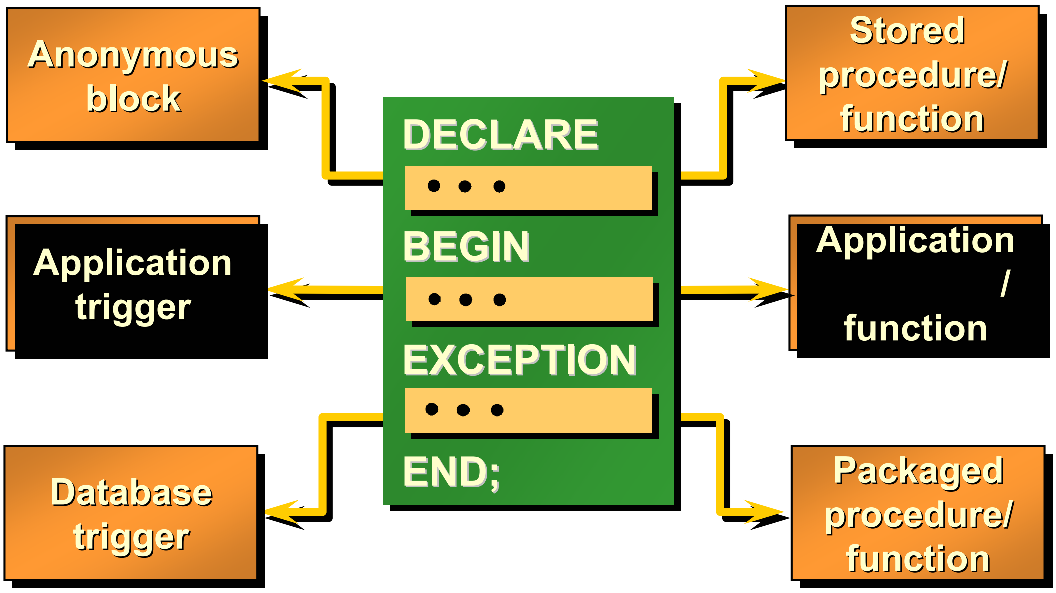
Procedure

```
PROCEDURE name
IS

BEGIN
   --statements

[EXCEPTION]

END;
```

Function

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

END;
```

Program Constructs



Use of Variables

- Use variables for:
  - Temporary storage of data
  - Manipulation of stored values
  - Reusability
  - Ease of maintenance

Handling Variables in PL/SQL

- Declare and initialize variables in the declaration section.
- Assign new values to variables in the executable section.
- Pass values into PL/SQL blocks through parameters.
- View results through output variables.

Types of Variables

- PL/SQL variables:
  - Scalar
  - Composite
  - Reference
  - LOB (large objects)
- Non-PL/SQL variables: Bind and host variables

Types of Variables

- PL/SQL variables:
  - Scalar
  - Composite
  - Reference
  - LOB (large objects)
- Non-PL/SQL variables: Bind and host variables

Declaring 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;
```

Declaring PL/SQL Variables

- Guidelines
  - Follow naming conventions.
  - Initialize variables designated as NOT NULL and CONSTANT.
  - Initialize identifiers by using the assignment operator (:=) or the DEFAULT reserved word.
  - Declare at most one identifier per line.

Naming Rules

– Two variables can have the same name, provided they are in different blocks.

– The variable name (identifier) should not be the same as the name of table columns used in the block.

```
DECLARE
  empno  NUMBER(4);
BEGIN
  SELECT    empno
  INTO      empno
  FROM      emp
  WHERE     ename = 'SMITH';
END;
```

Adopt a naming convention for PL/SQL identifiers: for example, v_empno

Assigning Values to Variables

Syntax

```
• identifier := expr;
```

Examples

Set a predefined hiredate for new employees.

```
v_hiredate := '31-DEC-98';
```

Set the employee name to Maduro.

```
v_ename := 'Maduro';
```

Variable Initialization and Keywords

- Using:
  - Assignment operator (:=)
  - DEFAULT keyword
  - NOT NULL constraint

Base Scalar Datatypes

- VARCHAR2 (*maximum_length*)
- NUMBER [(*precision, scale*)]
- DATE
- CHAR [(*maximum_length*)]
- LONG
- LONG RAW
- BOOLEAN
- BINARY_INTEGER
- PLS_INTEGER

Scalar Variable Declarations

- Examples

```
v_job           VARCHAR2(9);
v_count         BINARY_INTEGER := 0;
v_total_sal     NUMBER(9,2) := 0;
v_orderdate     DATE := SYSDATE + 7;
c_tax_rate      CONSTANT NUMBER(3,2) := 8.25;
v_valid         BOOLEAN NOT NULL := TRUE;
```

The %TYPE Attribute

– Declare a variable according to:
  - A database column definition
  - Another previously declared variable

– Prefix %TYPE with:
  - The database table and column
  - The previously declared variable name

Declaring Variables with the %TYPE Attribute

- Examples

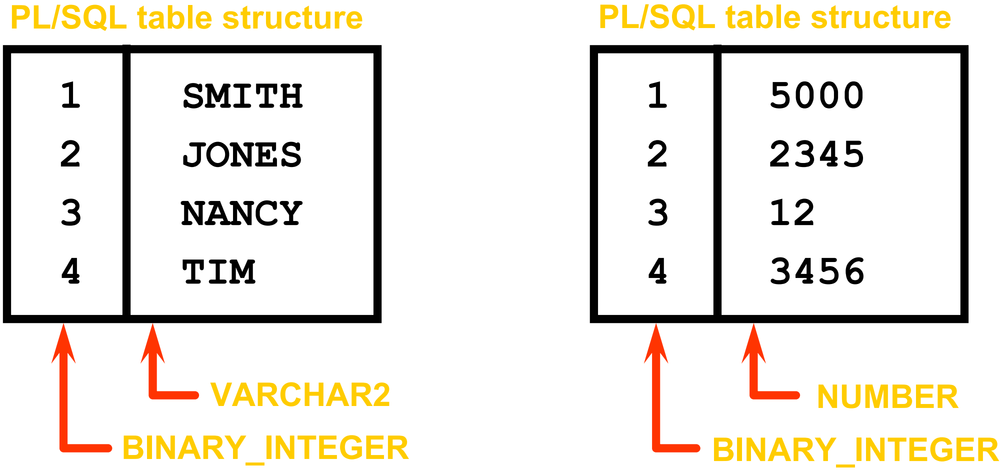
```
...
  v_ename               emp.ename%TYPE;
  v_balance             NUMBER(7,2);
  v_min_balance         v_balance%TYPE := 10;
...
```

Declaring Boolean Variables

- Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.
- The variables are connected by the logical operators AND, OR, and NOT.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

PL/SQL Record Structure

TRUE	23-DEC-98	ATLANTA	



PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

PL/SQL table structure

1	5000
2	2345
3	12
4	3456

DBMS_OUTPUT.PUT_LINE

- An Oracle-supplied packaged procedure
- An alternative for displaying data from a PL/SQL block
- Must be enabled in SQL*Plus with SET SERVEROUTPUT ON

Writing Executable Statements

Objectives

- After completing this lesson, you should be able to do the following:
  - Recognize the significance of the executable section
  - Write statements in the executable section
  - Describe the rules of nested blocks
  - Execute and test a PL/SQL block
  - Use coding conventions

PL/SQL Block Syntax and Guidelines

- Statements can continue over several lines.
- Lexical units can be separated by:
  - Spaces
  - Delimiters
  - Identifiers
  - Literals
  - Comments

PL/SQL Block Syntax and Guidelines

- Identifiers
  - Can contain up to 30 characters
  - Cannot contain reserved words unless enclosed in double quotation marks
  - Must begin with an alphabetic character
  - Should not have the same name as a database table column name

PL/SQL Block Syntax and Guidelines

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

```
v_ename := 'Henderson';
```

  - Numbers can be simple values or scientific notation.
- A PL/SQL block is terminated by a slash (/) on a line by itself.

Commenting Code

- Prefix single-line comments with two dashes (--).
- Place multi-line comments between the symbols /* and */.

- Example

```
...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_sal := &p_monthly_sal * 12;
END; -- This is the end of the block
```

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row number
  - Single-row character
  - Datatype conversion
  - Date

Same as in SQL

- Not available in procedural statements:
  - DECODE
  - Group functions

PL/SQL Functions

- Examples
  - Build the mailing list for a company.

```
v_mailing_address := v_name||CHR(10)||
                     v_address||CHR(10)||v_state||
                     CHR(10)||v_zip;
```

- Convert the employee name to lowercase.

```
v_ename          := LOWER(v_ename);
```

Datatype Conversion

- Convert data to comparable datatypes.
- Mixed datatypes can result in an error and affect performance.
- Conversion functions:
  - TO_CHAR
  - TO_DATE
  - TO_NUMBER

```
DECLARE
   v_date VARCHAR2(15);
BEGIN
   SELECT TO_CHAR(hiredate,
                  'MON. DD, YYYY')
   INTO   v_date
   FROM   emp
   WHERE  empno = 7839;
END;
```

Datatype Conversion

This statement produces a compilation error if the variable v_date is declared as datatype DATE.

```
v_date := 'January 13, 1998';
```

To correct the error, use the TO_DATE conversion function.

```
v_date := TO_DATE ('January 13, 1998',
                   'Month DD, YYYY');
```

Nested Blocks and Variable Scope

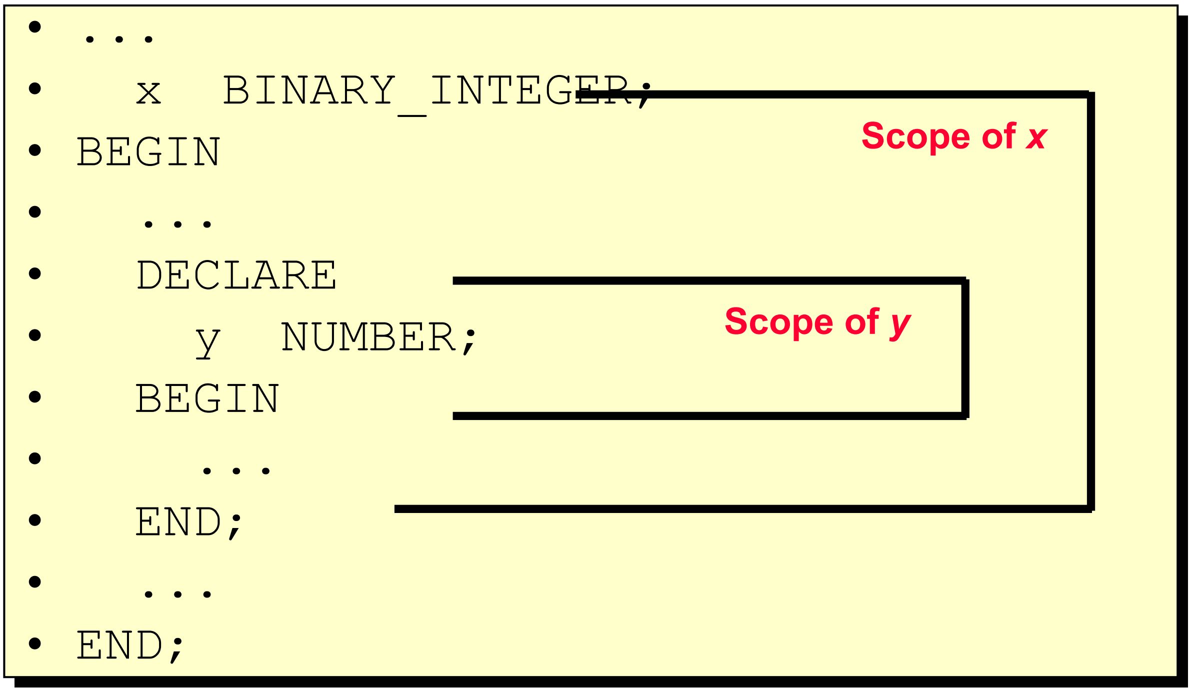
- Statements can be nested wherever an executable statement is allowed.
- A nested block becomes a statement.
- An exception section can contain nested blocks.
- The scope of an object is the region of the program that can refer to the object.

Nested Blocks and Variable Scope

- An identifier is visible in the regions in which you can reference the unqualified identifier:
  - A block can look up to the enclosing block.
  - A block cannot look down to enclosed blocks.

Nested Blocks and Variable Scope

Example



```
• ...
•    x  BINARY_INTEGER;
• BEGIN
•    ...
•    DECLARE
•       y  NUMBER;
•    BEGIN
•       ...
•    END;
•    ...
• END;
```

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations
- Exponential operator (**)

Same as in SQL

Operators in PL/SQL

- Examples
  - Increment the counter for a loop.

```
v_count       := v_count + 1;
```

  - Set the value of a Boolean flag.

```
v_equal       := (v_n1 = v_n2);
```

  - Validate an employee number if it contains a value.

```
v_valid       := (v_empno IS NOT NULL);
```

Code Naming Conventions

- Avoid ambiguity:
  - The names of local variables and formal parameters take precedence over the names of database tables.
  - The names of columns take precedence over the names of local variables.

Interacting with the Oracle Server

SQL Statements in PL/SQL

- Extract a row of data from the database by using the SELECT command. Only a single set of values can be returned.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the COMMIT, ROLLBACK, or SAVEPOINT command.
- Determine DML outcome with implicit cursors.

SELECT Statements in PL/SQL

- Retrieve data from the database with SELECT.
- Syntax

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

SELECT Statements in PL/SQL

- The INTO clause is required.
- Example

```
DECLARE
  v_deptno    NUMBER(2);
  v_loc       VARCHAR2(15);
BEGIN
  SELECT      deptno, loc
  INTO        v_deptno, v_loc
  FROM        dept
  WHERE       dname = 'SALES';
...
END;
```

Retrieving Data in PL/SQL

- Retrieve the order date and the ship date for the specified order.
- Example

```
DECLARE
  v_orderdate   ord.orderdate%TYPE;
  v_shipdate    ord.shipdate%TYPE;
BEGIN
  SELECT   orderdate, shipdate
  INTO     v_orderdate, v_shipdate
  FROM     ord
  WHERE    id = 620;
       ...
END;
```

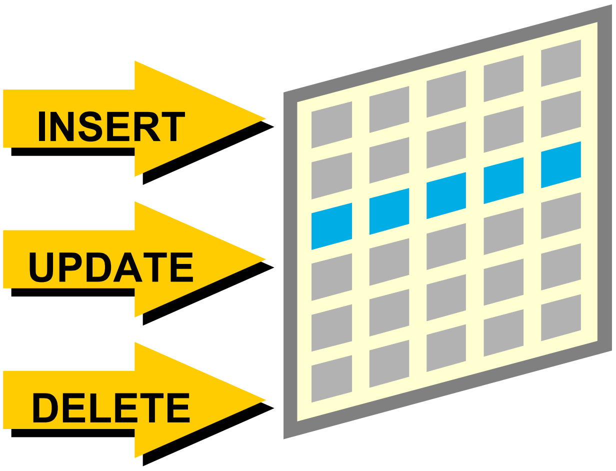
Retrieving Data in PL/SQL

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

```
DECLARE
  v_sum_sal    emp.sal%TYPE;
  v_deptno      NUMBER NOT NULL := 10;
BEGIN
  SELECT      SUM(sal)   -- group function
  INTO        v_sum_sal
  FROM        emp
  WHERE       deptno = v_deptno;
END;
```

Manipulating Data Using PL/SQL

- Make changes to database tables by using DML commands:
  - INSERT
  - UPDATE
  - DELETE



Inserting Data

- Add new employee information to the emp table.
- Example

```
BEGIN
    INSERT INTO   emp(empno, ename, job, deptno)
    VALUES        (empno_sequence.NEXTVAL, 'HARDING',
                  'CLERK', 10);
END;
```

Updating Data

- Increase the salary of all employees in the emp table who are Analysts.
- Example

```
DECLARE
  v_sal_increase   emp.sal%TYPE := 2000;
BEGIN
  UPDATE      emp
  SET         sal = sal + v_sal_increase
  WHERE       job = 'ANALYST';
END;
```

Deleting Data

- Delete rows that belong to department 10 from the emp table.
- Example

```
DECLARE
  v_deptno   emp.deptno%TYPE := 10;
BEGIN
  DELETE FROM   emp
  WHERE         deptno = v_deptno;
END;
```

Naming Conventions

– Use a naming convention to avoid ambiguity in the WHERE clause.

– Database columns and identifiers should have distinct names.

– Syntax errors can arise because PL/SQL checks the database first for a column in the table.

Naming Conventions

```
• DECLARE
•    orderdate    ord.orderdate%TYPE;
•   shipdate      ord.shipdate%TYPE;
•   ordid   ord.ordid%TYPE := 601;
• BEGIN
•    SELECT orderdate, shipdate
•   INTO    orderdate, shipdate
•   FROM    ord
•   WHERE   ordid = ordid;
• END;
• SQL> /
• DECLARE
• *
• ERROR at line 1:
• ORA-01422: exact fetch returns more than
  requested
• number of rows
• ORA-06512: at line 6
```

COMMIT and ROLLBACK Statements

- Initiate a transaction with the first DML command to follow a COMMIT or ROLLBACK.
- Use COMMIT and ROLLBACK SQL statements to terminate a transaction explicitly.

SQL Cursor

- A cursor is a private SQL work area.
- There are two types of cursors:
  - Implicit cursors
  - Explicit cursors
- The Oracle Server uses implicit cursors to parse and execute your SQL statements.
- Explicit cursors are explicitly declared by the programmer.

SQL Cursor Attributes

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

SQL%ROWCOUNT	**Number of rows affected by the most recent SQL statement (an integer value)**
SQL%FOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows**
SQL%NOTFOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows**
SQL%ISOPEN	**Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed**

SQL Cursor Attributes

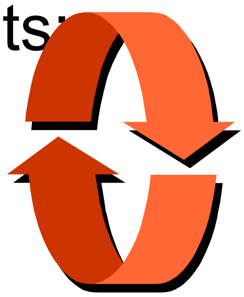
- Delete rows that have the specified order number from the ITEM table. Print the number of rows deleted.
- Example

```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
  v_ordid  NUMBER := 605;
BEGIN
  DELETE FROM  item
  WHERE        ordid = v_ordid;
  :rows_deleted := (SQL%ROWCOUNT ||
                       ' rows deleted.');
END;
/
PRINT rows_deleted
```

Writing Control Structures

Controlling PL/SQL Flow of Execution

- You can change the logical flow of statements using conditional IF statements and loop control structures.
- Conditional IF statements:
  - IF-THEN-END IF
  - IF-THEN-ELSE-END IF
  - IF-THEN-ELSIF-END IF



IF Statements

Syntax

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

Simple IF statement:

Set the manager ID to 22 if the employee name is Osborne.

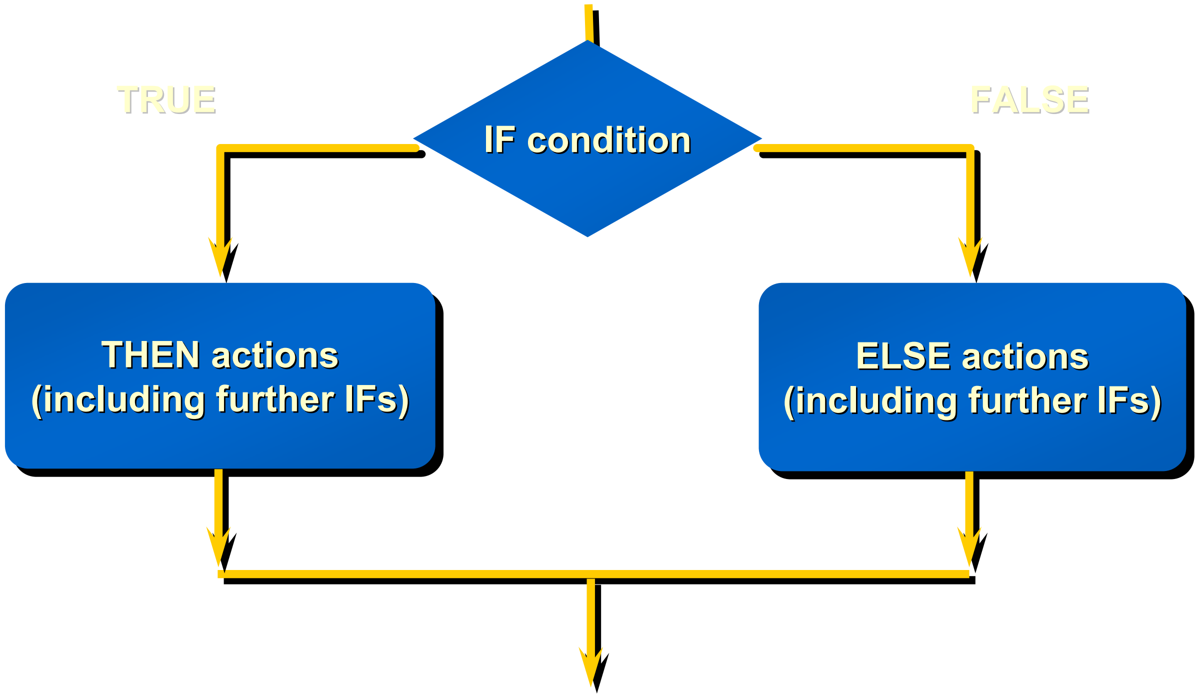
```
IF v_ename = 'OSBORNE' THEN
  v_mgr := 22;
END IF;
```

Simple IF Statements

- Set the job title to Salesman, the department number to 35, and the commission to 20% of the current salary if the last name is Miller.
- Example

```
. . .
IF v_ename      = 'MILLER' THEN
   v_job       := 'SALESMAN';
   v_deptno    := 35;
   v_new_comm  := sal * 0.20;
END IF;
. . .
```

IF-THEN-ELSE Statement Execution Flow

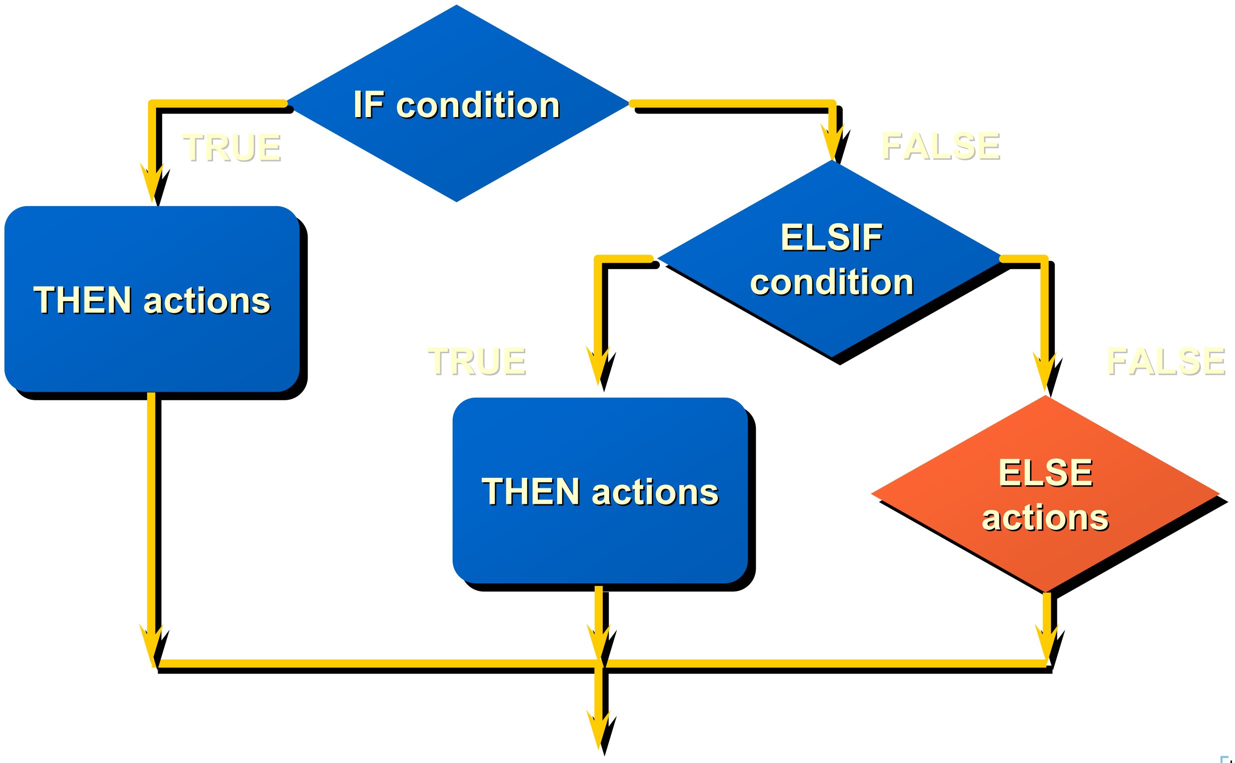


IF-THEN-ELSE Statements

- Set a flag for orders where there are fewer than five days between order date and ship date.
- Example

```
...
IF v_shipdate - v_orderdate < 5 THEN
  v_ship_flag := 'Acceptable';
ELSE
  v_ship_flag := 'Unacceptable';
END IF;
...
```

IF-THEN-ELSIF Statement Execution Flow



IF-THEN-ELSIF Statements

- For a given value, calculate a percentage of that value based on a condition.
- Example

```
. . .
IF    v_start > 100 THEN
      v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
      v_start := .5 * v_start;
ELSE
      v_start := .1 * v_start;
END IF;
. . .
```

Building Logical Conditions

- You can handle null values with the IS NULL operator.
- Any arithmetic expression containing a null value evaluates to NULL.
- Concatenated expressions with null values treat null values as an empty string.

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

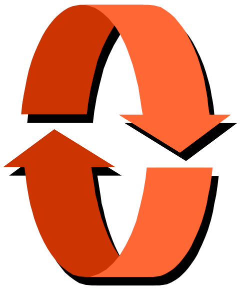
- What is the value of V_FLAG in each case?

```
v_flag := v_reorder_flag AND v_available_flag;
```

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
NULL	TRUE	NULL
NULL	FALSE	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 Loop

- Syntax

```
LOOP                                   -- delimiter
  statement1;                          -- statements
  . . .
  EXIT [WHEN condition];               -- EXIT statement
END LOOP;                              -- delimiter
```

```
where:   condition              is a Boolean variable or
                                expression (TRUE, FALSE,
                                or NULL);
```

Basic Loop

- Example

```
DECLARE
  v_ordid     item.ordid%TYPE := 601;
  v_counter   NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

FOR Loop

- Syntax

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

  - Use a FOR loop to shortcut the test for the number of iterations.
  - Do not declare the counter; it is declared implicitly.

FOR Loop

- Guidelines
  - Reference the counter within the loop only; it is undefined outside the loop.
  - Use an expression to reference the existing value of a counter.
  - Do *not* reference the counter as the target of an assignment.

FOR Loop

- Insert the first 10 new line items for order number 601.
- Example

```
DECLARE
  v_ordid     item.ordid%TYPE := 601;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, i);
  END LOOP;
END;
```

WHILE Loop

- Syntax

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

Condition is evaluated at the beginning of each iteration.

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

WHILE Loop

- Example

```
ACCEPT p_new_order PROMPT 'Enter the order number: '
ACCEPT p_items -
  PROMPT 'Enter the number of items in this order: '
DECLARE
v_count     NUMBER(2) := 1;
BEGIN
  WHILE v_count <= &p_items LOOP
    INSERT INTO item (ordid, itemid)
    VALUES (&p_new_order, v_count);
    v_count := v_count + 1;
  END LOOP;
  COMMIT;
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 referencing the label.

Nested Loops and Labels

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
  EXIT WHEN v_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;
```

Working with Composite Datatypes

Composite Datatypes

- Types:
  - PL/SQL RECORDS
  - PL/SQL TABLES
- Contain internal components
- Are reusable

PL/SQL Records

- Must contain one or more components of any scalar, RECORD, or PL/SQL TABLE datatype, called fields
- Are similar in structure to records in a 3GL
- Are not the same as rows in a database table
- Treat a collection of fields as a logical unit
- Are convenient for fetching a row of data from a table for processing

Creating a PL/SQL Record

- Syntax

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

- Where *field_declaration* is

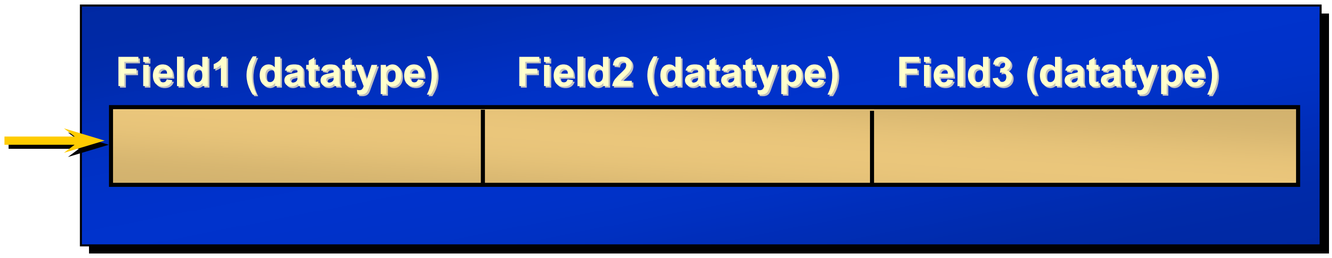
```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

Creating a PL/SQL Record

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

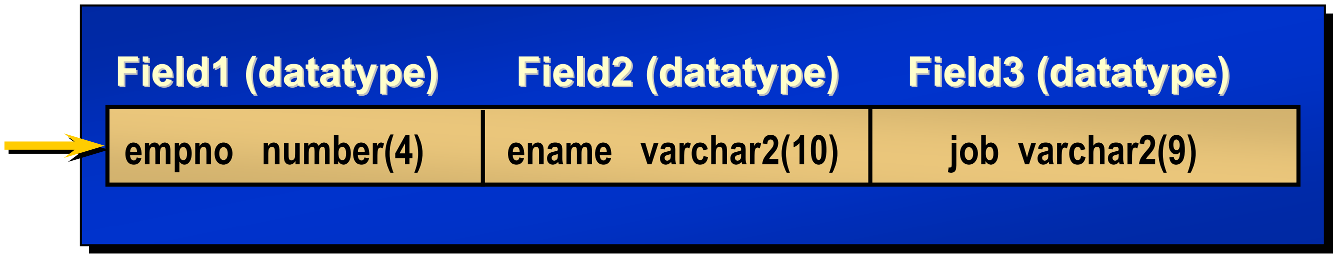
```
...
  TYPE emp_record_type IS RECORD
    (ename    VARCHAR2(10),
     job         VARCHAR2(9),
     sal         NUMBER(7,2));
  emp_record     emp_record_type;
...
```

PL/SQL Record Structure



Field1 (datatype) Field2 (datatype) Field3 (datatype)

Example



Field1 (datatype)	Field2 (datatype)	Field3 (datatype)
empno number(4)	ename varchar2(10)	job varchar2(9)

The %ROWTYPE Attribute

– Declare a variable according to a collection of columns in a database table or view.

– Prefix %ROWTYPE with the database table.

– Fields in the record take their names and datatypes from the columns of the table or view.

Advantages of Using %ROWTYPE

– The number and datatypes of the underlying database columns may not be known.

– The number and datatypes of the underlying database column may change at runtime.

– The attribute is useful when retrieving a row with the SELECT statement.

The %ROWTYPE Attribute

- Examples
- Declare a variable to store the same information about a department as it is stored in the DEPT table.

```
dept_record    dept%ROWTYPE;
```

- Declare a variable to store the same information about an employee as it is stored in the EMP table.

```
emp_record     emp%ROWTYPE;
```

PL/SQL Tables

– Are composed of two components:
  - Primary key of datatype BINARY_INTEGER
  - Column of scalar or record datatype

– Increase dynamically because they are unconstrained

Creating a PL/SQL Table

- Syntax

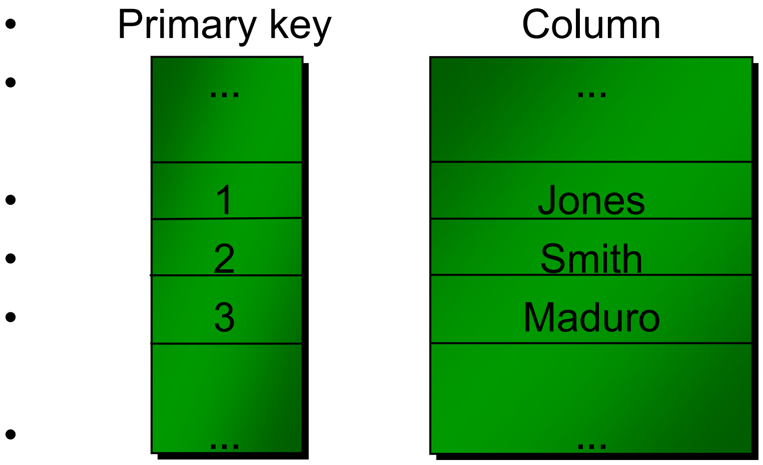
```
TYPE type_name IS TABLE OF
     {column_type | variable%TYPE
     | table.column%TYPE} [NOT NULL]
     [INDEX BY BINARY_INTEGER];
identifier   type_name;
```

Declare a PL/SQL table to store names.

Example

```
...
TYPE ename_table_type IS TABLE OF emp.ename%TYPE
  INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
...
```

PL/SQL Table Structure



- Primary key Column
- ... ...
- 1 Jones
- 2 Smith
- 3 Maduro
- ... ...
- BINARY_INTEGER Scalar

Creating a PL/SQL Table

```
• DECLARE
•   TYPE ename_table_type IS TABLE OF
  emp.ename%TYPE
•     INDEX BY BINARY_INTEGER;
•   TYPE hiredate_table_type IS TABLE OF DATE
•     INDEX BY BINARY_INTEGER;
•   ename_table     ename_table_type;
•   hiredate_table  hiredate_table_type;
• BEGIN
•   ename_table(1) := 'CAMERON';
•   hiredate_table(8) := SYSDATE + 7;
•     IF ename_table.EXISTS(1) THEN
•       INSERT INTO ...
•     ...
• END;
```

Using PL/SQL Table Methods

- The following methods make PL/SQL tables easier to use:
  - EXISTS
  - COUNT
  - FIRST and LAST
  - PRIOR
  - NEXT
  - EXTEND
  - TRIM
  - DELETE

PL/SQL Table of Records

- **Define a TABLE variable with a permitted PL/SQL datatype.**
- **Declare a PL/SQL variable to hold department information.**

Example

```
DECLARE
  TYPE dept_table_type IS TABLE OF dept%ROWTYPE
    INDEX BY BINARY_INTEGER;
  dept_table dept_table_type;
  -- Each element of dept_table is a record
```

Example of PL/SQL Table of Records

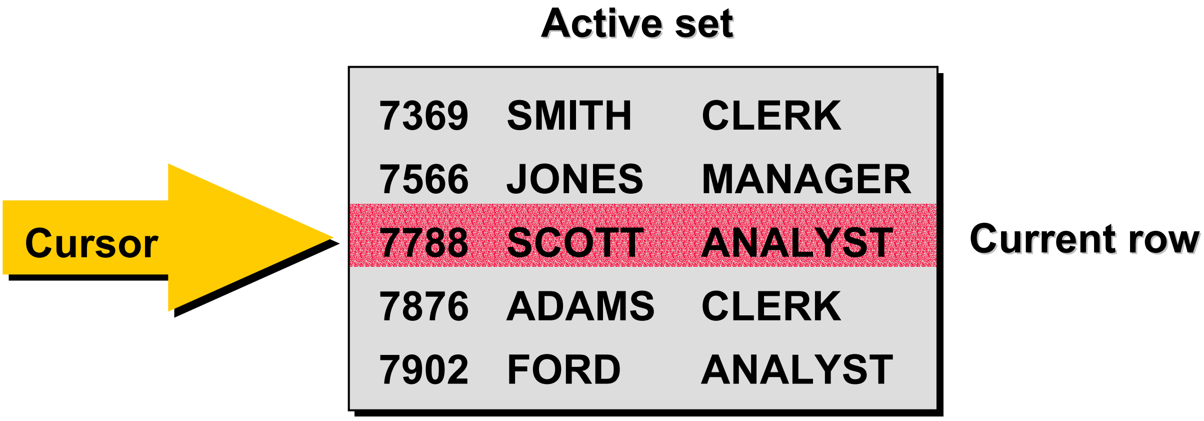
```
DECLARE
  TYPE e_table_type IS TABLE OF emp.Ename%Type
  INDEX BY BINARY_INTEGER;
  e_tab e_table_type;
BEGIN
  e_tab(1) := 'SMITH';
  UPDATE emp
  SET sal = 1.1 * sal
  WHERE Ename = e_tab(1);
  COMMIT;
END;
/
```

Writing Explicit Cursors

About Cursors

- Every SQL statement executed by the Oracle Server has an individual cursor associated with it:
  - Implicit cursors: Declared for all DML and PL/SQL SELECT statements
  - Explicit cursors: Declared and named by the programmer

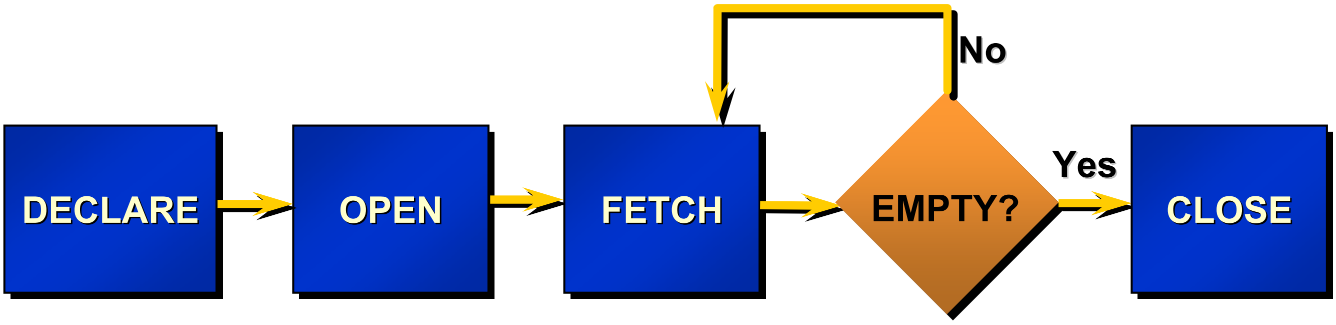
Explicit Cursor Functions



Active set

7369	SMITH	CLERK
7566	JONES	MANAGER
7788	SCOTT	ANALYST
7876	ADAMS	CLERK
7902	FORD	ANALYST

Controlling Explicit Cursors



DECLARE → OPEN → FETCH → EMPTY?

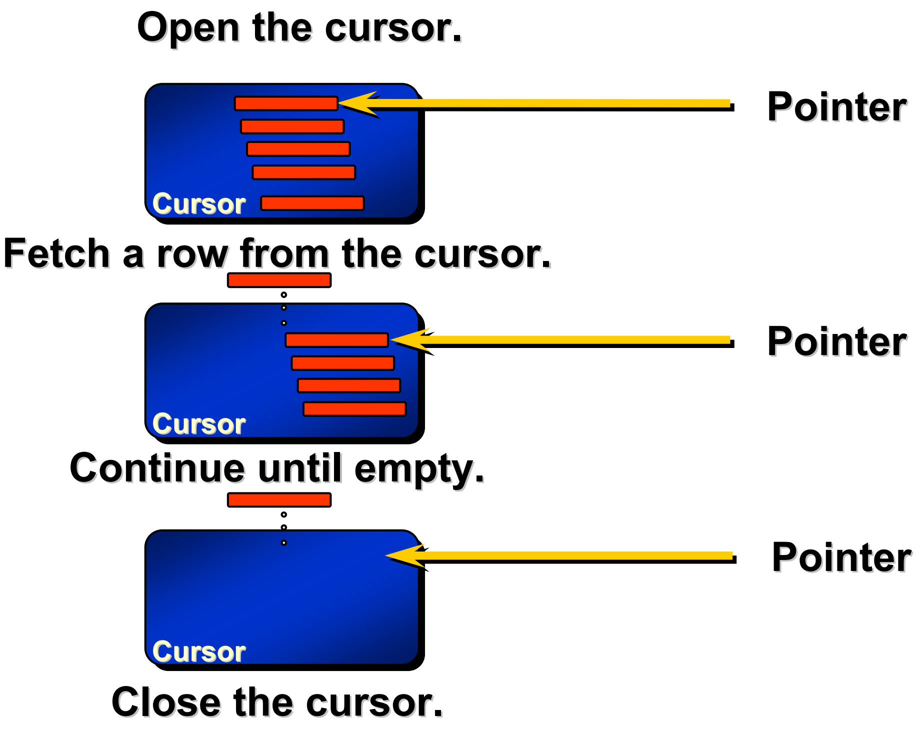
No

Yes

CLOSE

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

Controlling Explicit Cursors



Open the cursor.

Pointer

Cursor

Fetch a row from the cursor.

Pointer

Cursor

Continue until empty.

Pointer

Cursor

Close the cursor.

Declaring the Cursor

- Syntax

```
CURSOR cursor_name IS
     select_statement;
```

  - Do not include the INTO clause in the cursor declaration.
  - If processing rows in a specific sequence is required, use the ORDER BY clause in the query.

Declaring the Cursor

- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename
    FROM   emp;

  CURSOR dept_cursor IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
  ...
```

Opening the Cursor

- Syntax

```
OPEN  cursor_name;
```

  - Open the cursor to execute the query and identify the active set.
  - If the query returns no rows, no exception is raised.
  - Use cursor attributes to test the outcome after a fetch.

Fetching Data from the Cursor

- Syntax

```
FETCH cursor_name INTO  [variable1, variable2, ...]
                        | record_name];
```

  - Retrieve the current row values into variables.
  - Include the same number of variables.
  - Match each variable to correspond to the columns positionally.
  - Test to see if the cursor contains rows.

Fetching Data from the Cursor

- Examples

```
FETCH emp_cursor INTO v_empno, v_ename;
```

-

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```

Closing the Cursor

- Syntax

```
CLOSE    cursor_name;
```

  - Close the cursor after completing the processing of the rows.
  - Reopen the cursor, if required.
  - Do not attempt to fetch data from a cursor once it has been closed.

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

Controlling Multiple Fetches

- – Process several rows from an explicit cursor using a loop.
- – Fetch a row with each iteration.
- – Use the %NOTFOUND attribute to write a test for an unsuccessful fetch.
- – Use explicit cursor attributes to test the success of each fetch.

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...
```

The %NOTFOUND and %ROWCOUNT Attributes

- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows.
- Use the %NOTFOUND cursor attribute to determine when to exit the loop.

Cursors and Records

- Process the rows of the active set conveniently by fetching values into a PL/SQL RECORD.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT  empno, ename
    FROM    emp;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
  ...
```

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, and close occur.
- The record is implicitly declared.

Cursor FOR Loops

- Retrieve employees one by one until no more are left.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM   emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Cursor FOR Loops Using Subqueries

- No need to declare the cursor.
- Example

```
BEGIN
  FOR emp_record IN (SELECT ename, deptno
                     FROM   emp) LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Advanced Explicit Cursor Concepts

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.

Cursors with Parameters

- Pass the department number and job title to the WHERE clause.
- Example

```
DECLARE
  CURSOR emp_cursor
  (p_deptno NUMBER, p_job VARCHAR2) IS
    SELECT    empno, ename
    FROM      emp
    WHERE     deptno = v_deptno
     AND      job = v_job;
BEGIN
  OPEN emp_cursor(10, 'CLERK');
...
```

The FOR UPDATE Clause

- Syntax

```
SELECT ...
FROM          ...
FOR UPDATE [OF column_reference][NOWAIT];
```

  – Explicit locking lets you deny access for the duration of a transaction.

  – Lock the rows *before* the update or delete.

The FOR UPDATE Clause

- Retrieve the employees who work in department 30.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename, sal
    FROM   emp
    WHERE  deptno = 30
    FOR UPDATE OF sal NOWAIT;
```

The WHERE CURRENT OF Clause

- Syntax

```
 WHERE CURRENT OF cursor ;
```

  - Use cursors to update or delete the current row.
  - Include the FOR UPDATE clause in the cursor query to lock the rows first.
  - Use the WHERE CURRENT OF clause to reference the current row from an explicit cursor.

The WHERE CURRENT OF Clause

Example

```
•  DECLARE
•    CURSOR sal_cursor IS
•      SELECT      sal
•      FROM  emp
•      WHERE deptno = 30
•      FOR UPDATE OF sal NOWAIT;
•  BEGIN
•    FOR emp_record IN sal_cursor LOOP
•      UPDATE      emp
•      SET   sal = emp_record.sal * 1.10
•      WHERE CURRENT OF sal_cursor;
•    END LOOP;
•    COMMIT;
•  END;
```

Cursors with Subqueries

Example

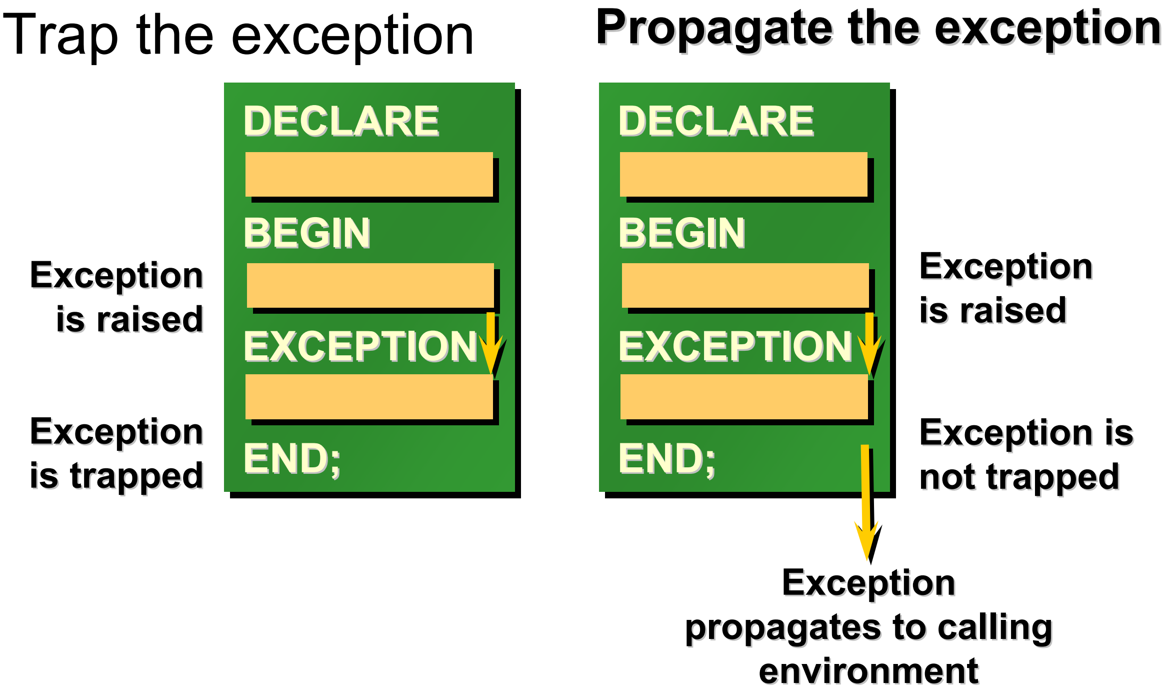
```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.deptno, t1.dname, t2.STAFF
    FROM   dept t1, (SELECT deptno,
                            count(*) STAFF
                     FROM   emp
                     GROUP BY deptno) t2
    WHERE  t1.deptno = t2.deptno
    AND    t2.STAFF >= 5;
```

Handling Exceptions

Handling Exceptions with PL/SQL

- What is an exception?
  Identifier in PL/SQL that is raised during execution
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

Handling Exceptions



Trap the exception

DECLARE

BEGIN

EXCEPTION

END;

Exception is raised

Exception is trapped

Propagate the exception

DECLARE

BEGIN

EXCEPTION

END;

Exception is raised

Exception is not trapped

Exception propagates to calling environment

Exception Types

- – Predefined Oracle Server
- – Non-predefined Oracle Server
- – User-defined

} **Implicitly raised**

Explicitly raised

Trapping Exceptions

- Syntax

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

Trapping Exceptions Guidelines

– WHEN OTHERS is the last clause.

– EXCEPTION keyword starts exception-handling section.

– Several exception handlers are allowed.

– Only one handler is processed before leaving the block.

Trapping Predefined Oracle Server Errors

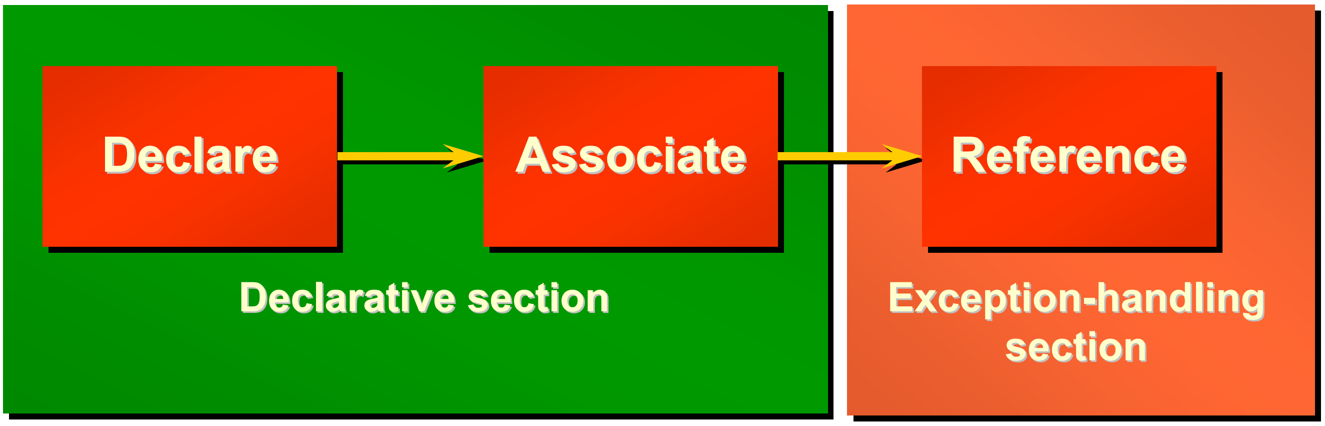
- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

Predefined Exception

- Syntax

```
BEGIN
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

Trapping Non-Predefined Oracle Server Errors



- Name the exception
- Code the PRAGMA EXCEPTION_INIT
- Handle the raised exception

Non-Predefined Error

- Trap for Oracle Server error number –2292, an integrity constraint violation.

```
DECLARE
   e_emps_remaining      EXCEPTION;                  1
   PRAGMA EXCEPTION_INIT (
                e_emps_remaining, -2292);            2
  v_deptno      dept.deptno%TYPE := &p_deptno;
BEGIN
  DELETE FROM dept
  WHERE         deptno = v_deptno;
  COMMIT;
EXCEPTION
  WHEN e_emps_remaining THEN                         3
   DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
   TO_CHAR(v_deptno) || '.  Employees exist. ');
END;
```

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

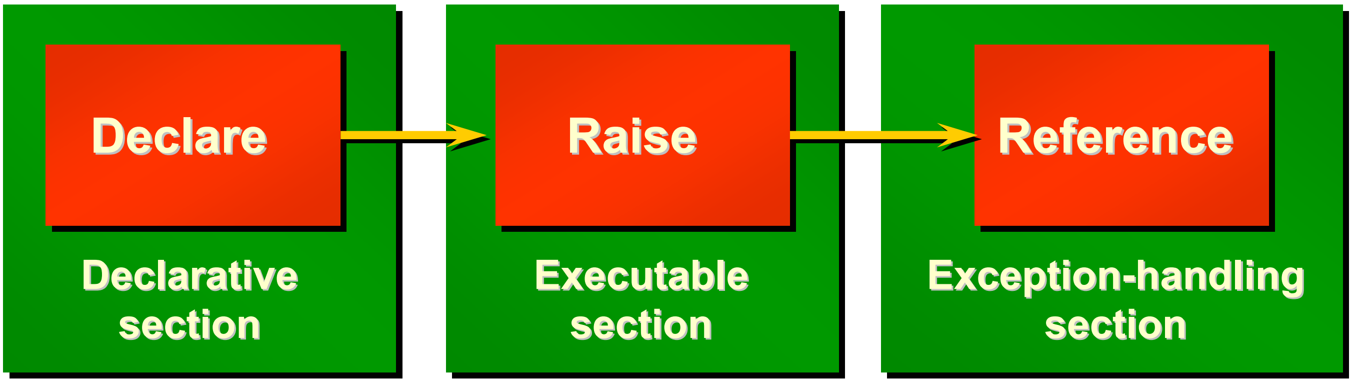
- Example

```
DECLARE
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQI
    v_error_message := SQLERRM ;
    INSERT INTO errors VALUES(v_error_code,
                              v_error_message);
END;
```

SQLCODE

SQLERRM

Trapping User-Defined Exceptions



- Name the exception
- Explicitly raise the exception by using the RAISE statement
- Handle the raised exception

User-Defined Exception

Example

```
DECLARE
  e_invalid_product  EXCEPTION;                                1
BEGIN
  UPDATE      product
  SET         descrip = '&product_description'
  WHERE       prodid = &product_number;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_product;                                   2
  END IF;
  COMMIT;
EXCEPTION
  WHEN  e_invalid_product  THEN                                3
    DBMS_OUTPUT.PUT_LINE('Invalid product number.');
END;
```

Propagating Exceptions

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
  . . .
  e_no_rows       exception;
  e_integrity     exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP
    BEGIN
      SELECT ...
      UPDATE ...
      IF SQL%NOTFOUND THEN
        RAISE e_no_rows;
      END IF;
    EXCEPTION
      WHEN e_integrity THEN ...
      WHEN e_no_rows THEN ...
    END;
END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN . . .
  WHEN TOO_MANY_ROWS THEN . . .
END;
```

RAISE_APPLICATION_ERROR Procedure

- Syntax

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

  - A procedure that lets you issue user-defined error messages from stored subprograms
  - Called only from an executing stored subprogram

RAISE_APPLICATION_ERROR Procedure

- Used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle Server errors