I

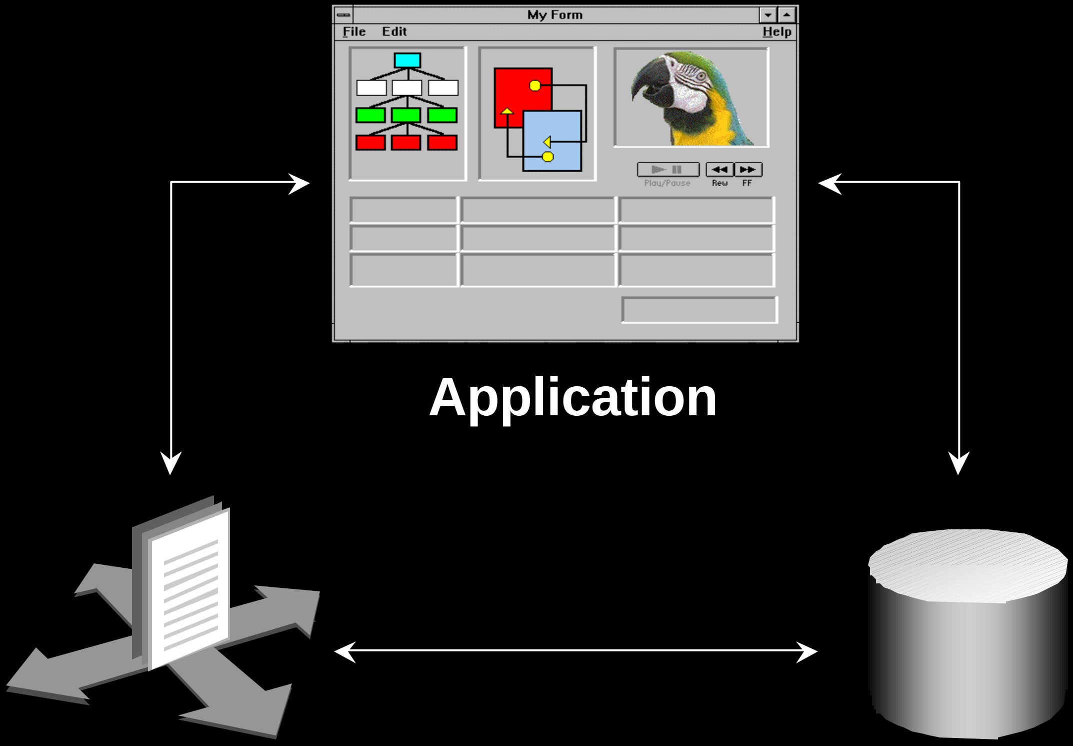
Overview of PL/SQL

About PL/SQL

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

Benefits of PL/SQL

Integration



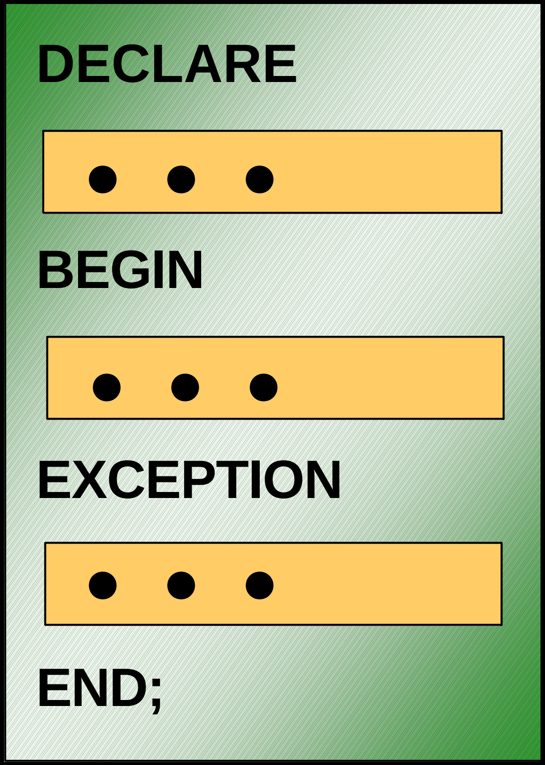
Application

Shared library

Oracle server

Benefits of PL/SQL

Modularize program development



DECLARE

● ● ●

BEGIN

● ● ●

EXCEPTION

● ● ●

END;

Benefits of PL/SQL

- PL/SQL is portable.
- You can declare variables.

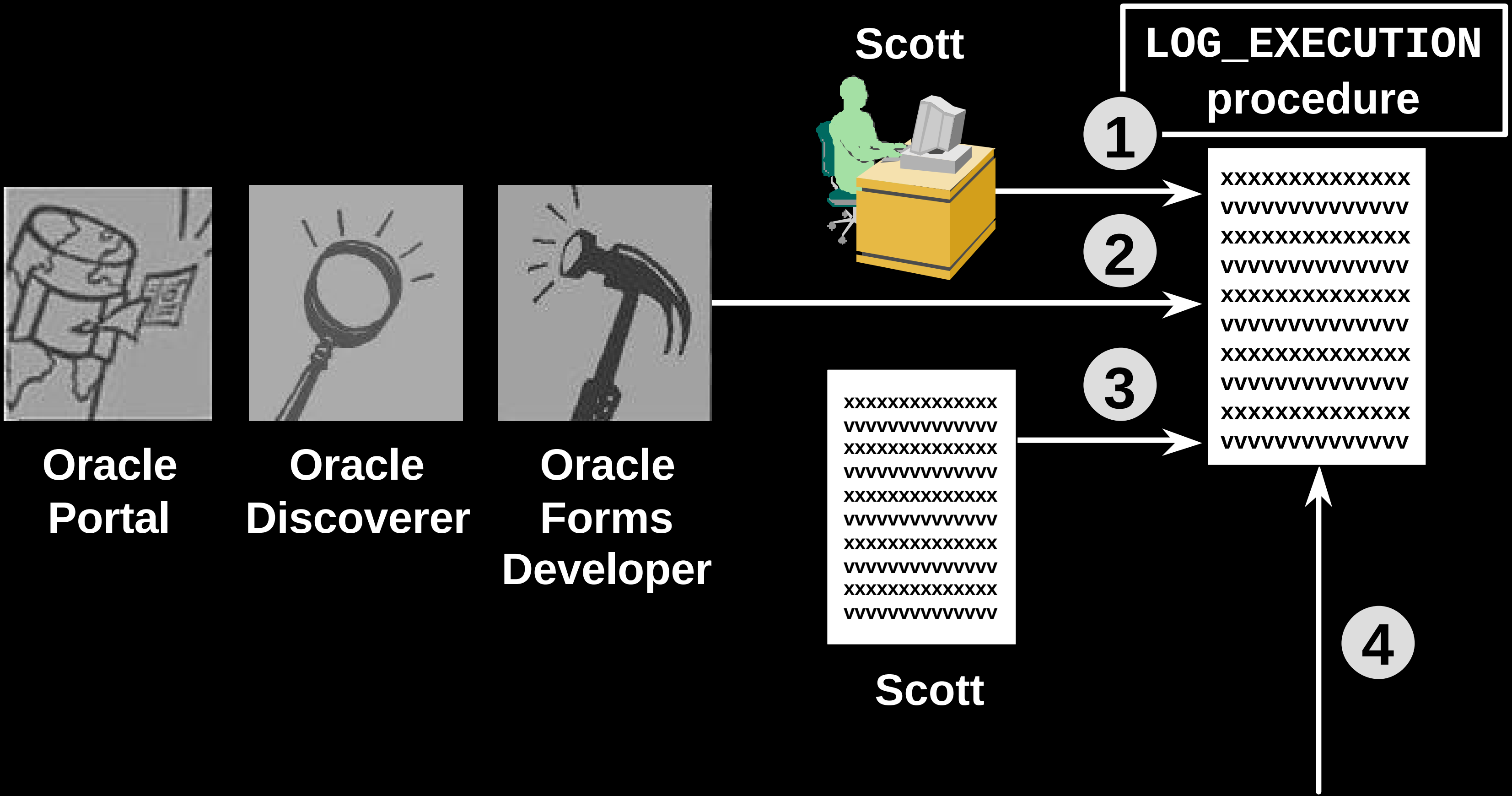
Benefits of PL/SQL

- You can program with procedural language control structures.
- PL/SQL can handle errors.

Benefits of Subprograms

- Easy maintenance
- Improved data security and integrity
- Improved performance
- Improved code clarity

Invoking Stored Procedures and Functions



Scott

LOG_EXECUTION procedure

1

2

3

4

Oracle Portal

Oracle Discoverer

Oracle Forms Developer

Scott

1

Declaring Variables

Objectives

After completing this lesson, you should be able to do the following:

- Recognize the basic PL/SQL block and its sections
- Describe the significance of variables in PL/SQL
- Declare PL/SQL variables
- Execute a PL/SQL block

PL/SQL Block Structure

`DECLARE` (Optional)

Variables, cursors, user-defined exceptions

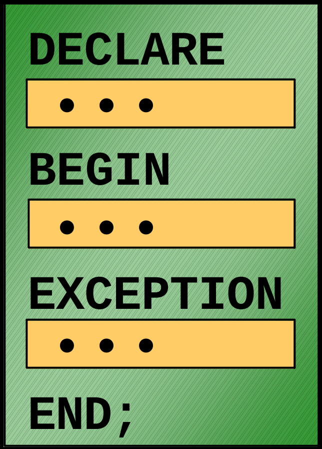
`BEGIN` (Mandatory)

- SQL statements
- PL/SQL statements

`EXCEPTION` (Optional)

Actions to perform when errors occur

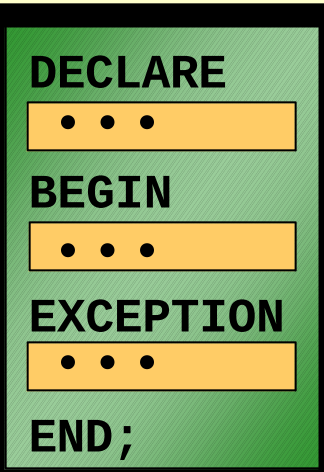
`END;` (Mandatory)



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

Executing Statements and PL/SQL Blocks

```
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;
```

Use of Variables

Variables can be used 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**

Using *i*SQL*Plus Variables Within PL/SQL Blocks

- PL/SQL does not have input or output capability of its own.
- You can reference substitution variables within a PL/SQL block with a preceding ampersand.
- *i*SQL*Plus host (or “bind”) variables can be used to pass run time values out of the PL/SQL block back to the *i*SQL*Plus environment.

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;
```

Guidelines for Declaring PL/SQL Variables

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

```
identifier := expr;
```

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

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

Variable Initialization and Keywords

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

Syntax:

```
identifier := expr;
```

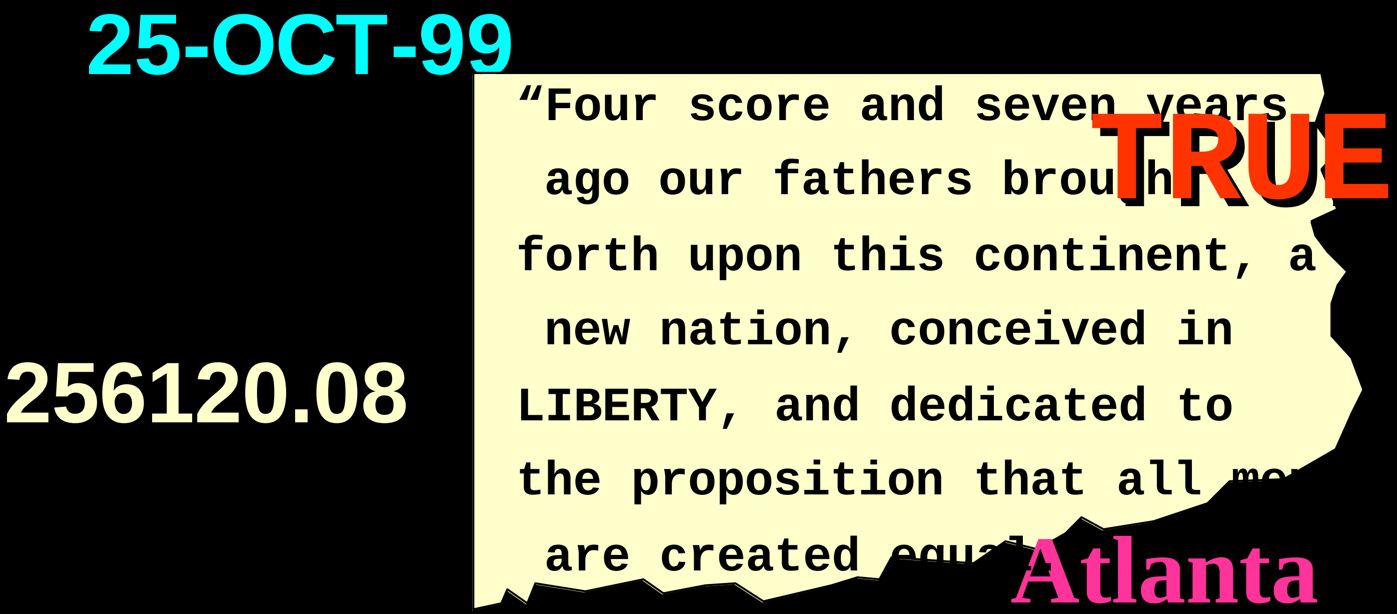
Examples:

```
v_hiredate := '01-JAN-2001';
```

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

Scalar Data Types

- Hold a single value
- Have no internal components



25-OCT-99

“Four score and seven years
ago our fathers brought
forth upon this continent, a
new nation, conceived in
LIBERTY, and dedicated to
the proposition that all men
are created equal.

TRUE

256120.08

Atlanta

Base Scalar Data Types

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

Scalar Variable Declarations

Examples:

```
DECLARE
  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

ORACLE

Declaring Variables with the %TYPE Attribute

Syntax:

```
identifier     Table.column_name%TYPE;
```

Examples:

```
...
  v_name            employees.last_name%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 compared 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.

Composite Data Types

TRUE	23-DEC-98	ATLANTA	

PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

VARCHAR2

BINARY_INTEGER

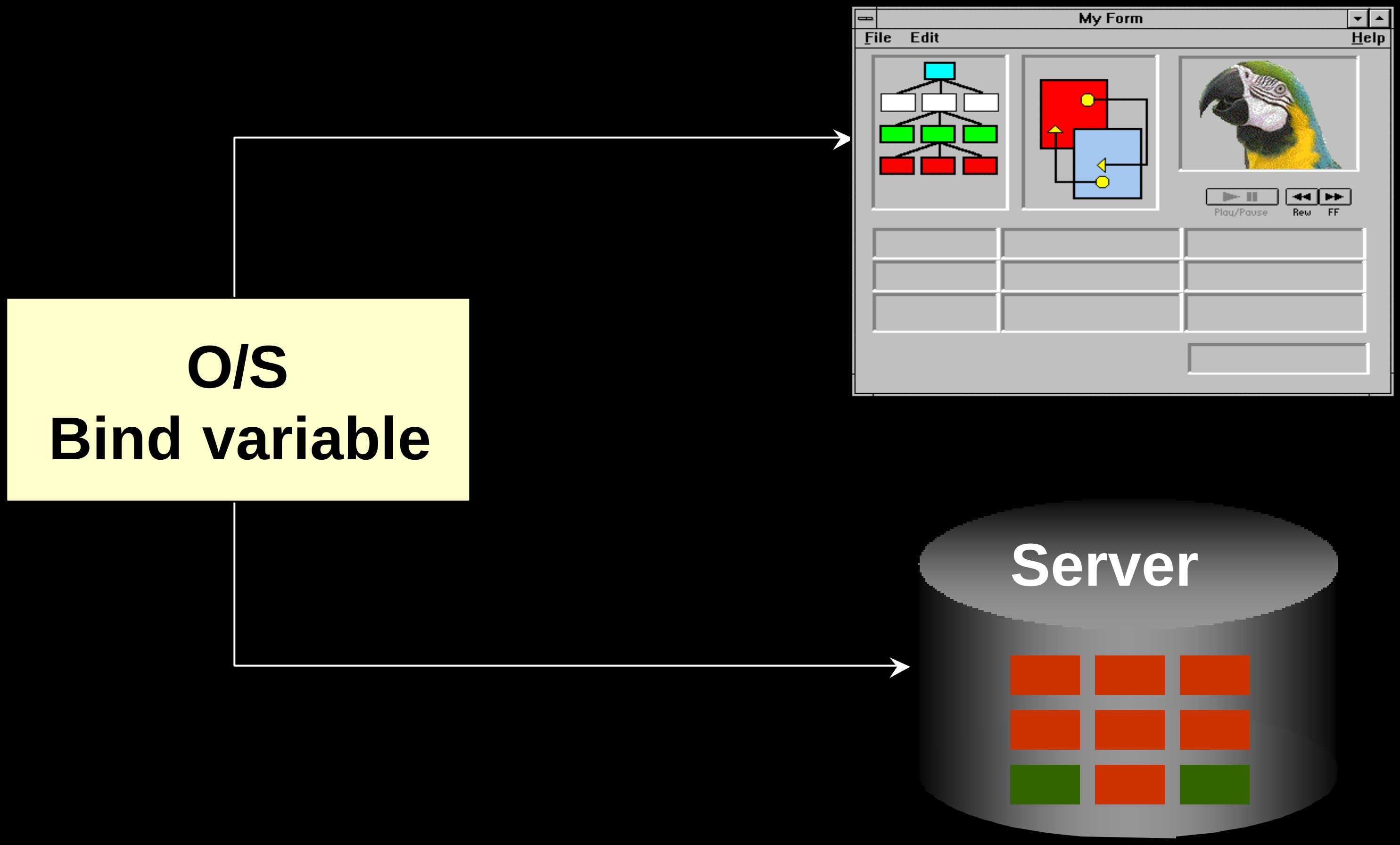
PL/SQL table structure

1	5000
2	2345
3	12
4	3456

NUMBER

BINARY_INTEGER

Bind Variables



O/S
Bind variable

Server

Using Bind Variables

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).

Example:

```
VARIABLE      g_salary NUMBER
BEGIN
   SELECT     salary
   INTO       :g_salary
   FROM       employees
   WHERE      employee_id = 178;
END;
/
PRINT g_salary
```

Referencing Non-PL/SQL Variables

Store the annual salary into a *i*SQL*Plus host variable.

```
:g_monthly_sal := v_sal / 12;
```

- **Reference non-PL/SQL variables as host variables.**
- **Prefix the references with a colon (:).**

DBMS_OUTPUT.PUT_LINE

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

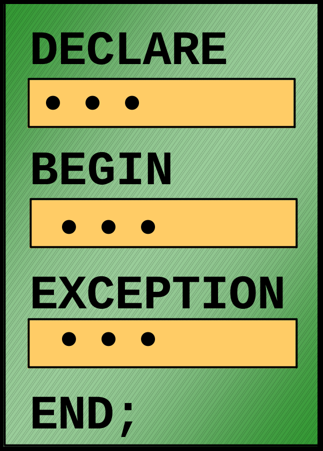
```
SET SERVEROUTPUT ON
DEFINE p_annual_sal = 60000
```

```
DECLARE
  v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
  v_sal := v_sal/12;
  DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' ||
                         TO_CHAR(v_sal));
END;
/
```

Summary

In this lesson you should have learned that:

- PL/SQL blocks are composed of the following sections:
  - Declarative (optional)
  - Executable (required)
  - Exception handling (optional)
- A PL/SQL block can be an anonymous block, procedure, or function.



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

Summary

In this lesson you should have learned that:

- PL/SQL identifiers:
  - Are defined in the declarative section
  - Can be of scalar, composite, reference, or `LOB` data type
  - Can be based on the structure of another variable or database object
  - Can be initialized
- Variables declared in an external environment such as *i*SQL*Plus are called host variables.
- Use `DBMS_OUTPUT.PUT_LINE` to display data from a PL/SQL block.

Practice 1 Overview

This practice covers the following topics:

- Determining validity of declarations
- Declaring a simple PL/SQL block
- Executing a simple PL/SQL block

2

Writing Executable Statements

Objectives

After completing this lesson, you should be able to do the following:

- Describe the significance of the executable section
- Use identifiers correctly
- 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 classified as:
  - Delimiters
  - Identifiers
  - Literals
  - Comments

Identifiers

- Can contain up to 30 characters
- Must begin with an alphabetic character
- Can contain numerals, dollar signs, underscores, and number signs
- Cannot contain characters such as hyphens, slashes, and spaces
- Should not have the same name as a database table column name
- Should not be reserved words

PL/SQL Block Syntax and Guidelines

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

```
v_name := 'Henderson';
```

  - Numbers can be simple values or scientific notation.
- A slash (/) runs the PL/SQL block in a script file or in some tools such as *i*SQL*PLUS.

Commenting Code

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

Example:

```
DECLARE
...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_sal := :g_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
  - Data type conversion
  - Date
  - Timestamp
  - GREATEST and LEAST
  - Miscellaneous functions

Same as in SQL

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

Data Type Conversion

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

```
DECLARE
   v_date DATE := TO_DATE('12-JAN-2001', 'DD-MON-YYYY');
BEGIN
    . . .
```

Data Type Conversion

This statement produces a compilation error if the variable `v_date` is declared as a DATE data type.

```
v_date := 'January 13, 2001';
```

Data Type Conversion

To correct the error, use the TO_DATE conversion function.

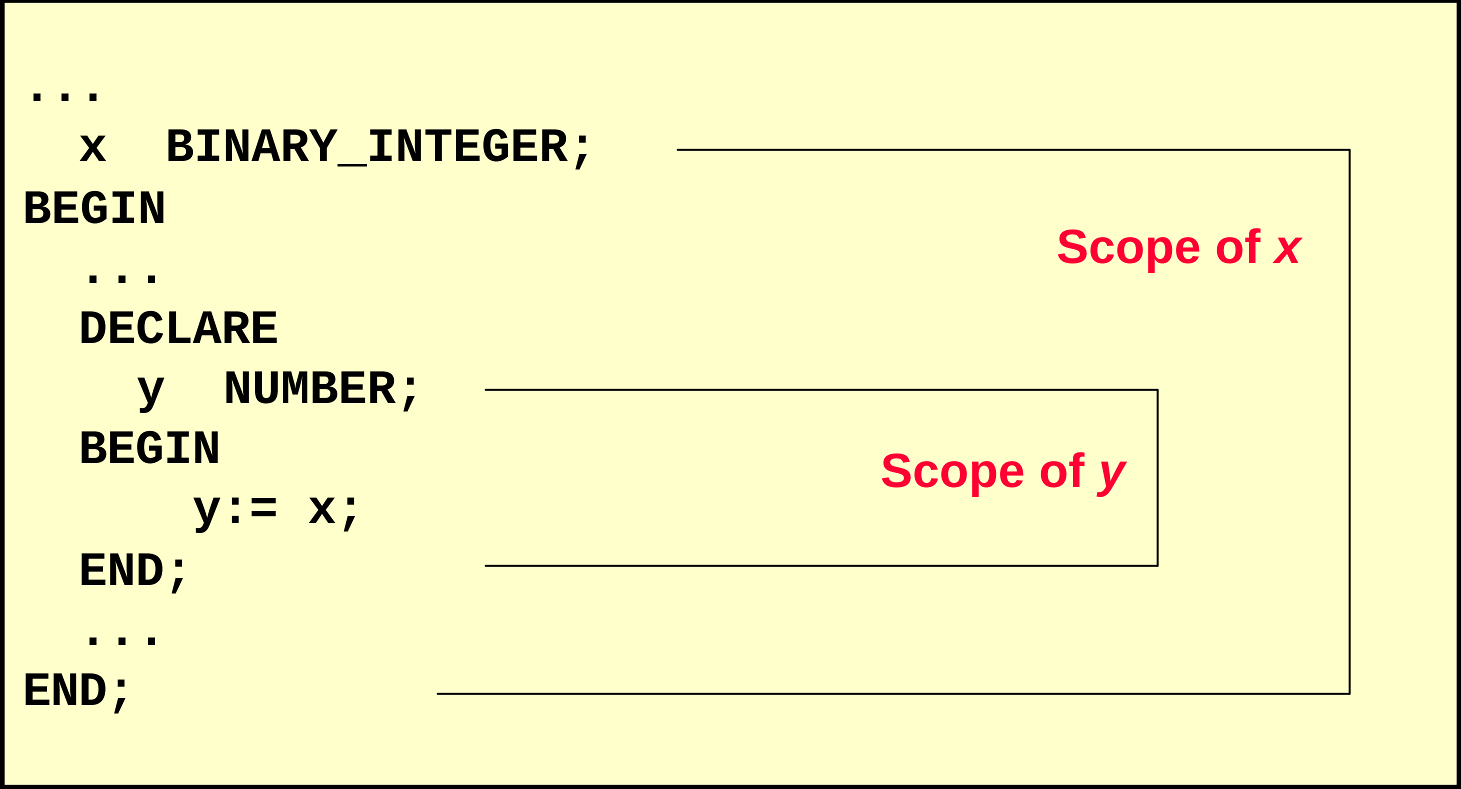
```
v_date := TO_DATE ('January 13, 2001',
                   'Month DD, YYYY');
```

Nested Blocks
and Variable Scope

- PL/SQL blocks 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 identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.

Nested Blocks and Variable Scope

Example:



```
...
  x  BINARY_INTEGER;
BEGIN
  ...
  DECLARE
    y  NUMBER;
  BEGIN
      y:= x;
  END;
  ...
END;
```

Scope of *x*

Scope of *y*

Operators in PL/SQL

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

} Same as in SQL

- Exponential operator (**)

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 whether an employee number contains a value.**

```
v_valid          := (v_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:

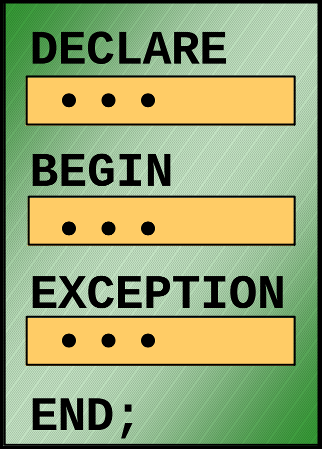
```
BEGIN
  IF x=0 THEN
     y:=1;
  END IF;
END;
```

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

Summary

In this lesson you should have learned that:

- PL/SQL block syntax and guidelines
- How to use identifiers correctly
- PL/SQL block structure: nesting blocks and scoping rules
- PL/SQL programming:
  - Functions
  - Data type conversions
  - Operators
  - Conventions and guidelines



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

Practice 2 Overview

This practice covers the following topics:

- Reviewing scoping and nesting rules
- Developing and testing PL/SQL blocks

3

Interacting with the Oracle Server

Objectives

After completing this lesson, you should be able to do the following:

- Write a successful SELECT statement in PL/SQL
- Write DML statements in PL/SQL
- Control transactions in PL/SQL
- Determine the outcome of SQL data manipulation language (DML) statements

SQL Statements in PL/SQL

- Extract a row of data from the database by using the `SELECT` command.
- 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 cursor attributes.

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];
```

SELECT Statements in PL/SQL

- The `INTO` clause is required.
- Queries must return one and only one row.

Example:

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

Retrieving Data in PL/SQL

Retrieve the hire date and the salary for the specified employee.

Example:

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

Retrieving Data in PL/SQL

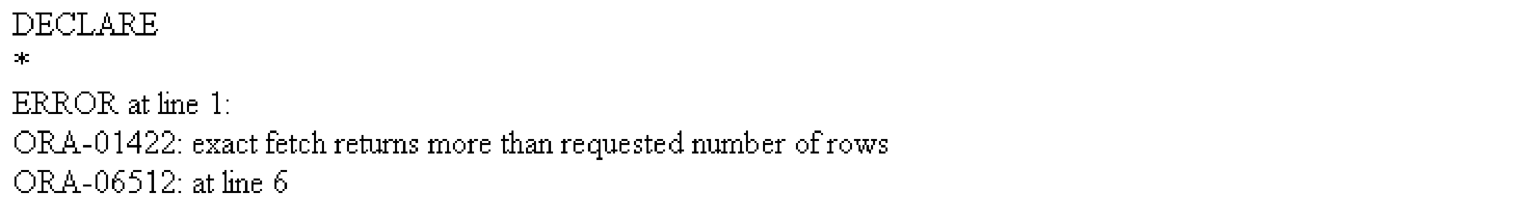
Return the sum of the salaries for all employees in the specified department.

Example:

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

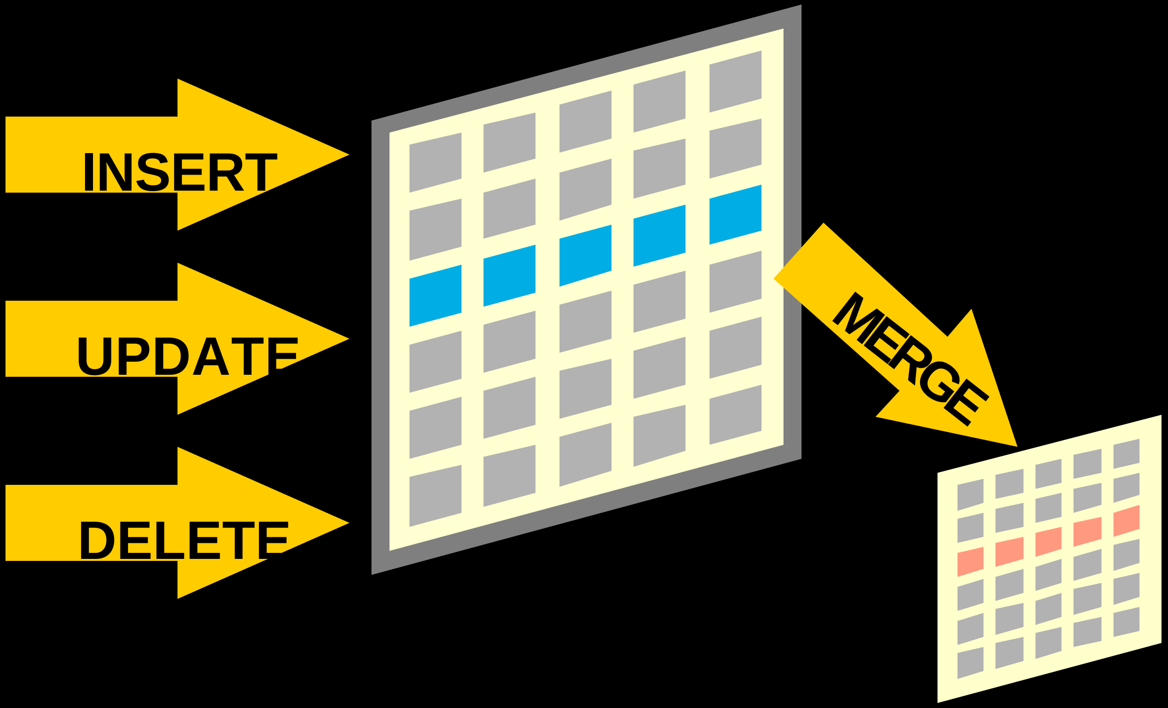


```
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
```

Manipulating Data Using PL/SQL

Make changes to database tables by using DML commands:

- INSERT
- UPDATE
- DELETE
- MERGE



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

Deleting Data

Delete rows that belong to department 10 from the EMPLOYEES table.

Example:

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

Merging Rows

Insert or update rows in the `COPY_EMP` table to match the `EMPLOYEES` table.

```
DECLARE
      v_empno employees.employee_id%TYPE := 100;
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = v_empno)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name     = e.first_name,
       c.last_name      = e.last_name,
       c.email          = e.email,
       . . .
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id, e.first_name, e.last_name,
          . . .,e.department_id);
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.
- The names of local variables and formal parameters take precedence over the names of database tables.
- The names of database table columns take precedence over the names of local variables.

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 statements.

`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 employee ID from the EMPLOYEES table. Print the number of rows deleted.

Example:

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

Transaction Control 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.**

Summary

In this lesson you should have learned how to:

- Embed SQL in the PL/SQL block using `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `MERGE`
- Embed transaction control statements in a PL/SQL block `COMMIT`, `ROLLBACK`, and `SAVEPOINT`

Summary

In this lesson you should have learned that:

- There are two cursor types: implicit and explicit.
- Implicit cursor attributes are used to verify the outcome of DML statements:
  - `SQL%ROWCOUNT`
  - `SQL%FOUND`
  - `SQL%NOTFOUND`
  - `SQL%ISOPEN`
- Explicit cursors are defined by the programmer.

Practice 3 Overview

This practice covers creating a PL/SQL block to:

- Select data from a table
- Insert data into a table
- Update data in a table
- Delete a record from a table

4

Writing Control Structures

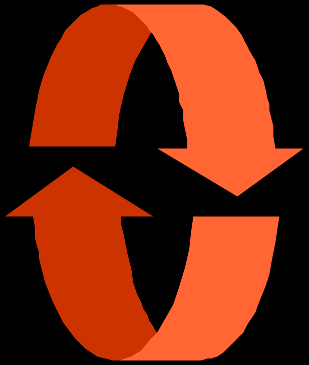
Objectives

After completing this lesson, you should be able to do the following:

- Identify the uses and types of control structures
- Construct an `IF` statement
- Use CASE expressions
- Construct and identify different loop statements
- Use logic tables
- Control block flow using nested loops and labels

Controlling PL/SQL Flow of Execution

- You can change the logical execution 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`



ORACLE

IF Statements

Syntax:

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

If the employee name is Gietz, set the Manager ID to 102.

```
IF UPPER(v_last_name) = 'GIETZ' THEN
  v_mgr := 102;
END IF;
```

Simple IF Statements

If the last name is Vargas:

- Set job ID to SA_REP
- Set department number to 80

```
. . .
IF v_ename      = 'Vargas' THEN
   v_job       := 'SA_REP';
   v_deptno    := 80;
END IF;
. . .
```

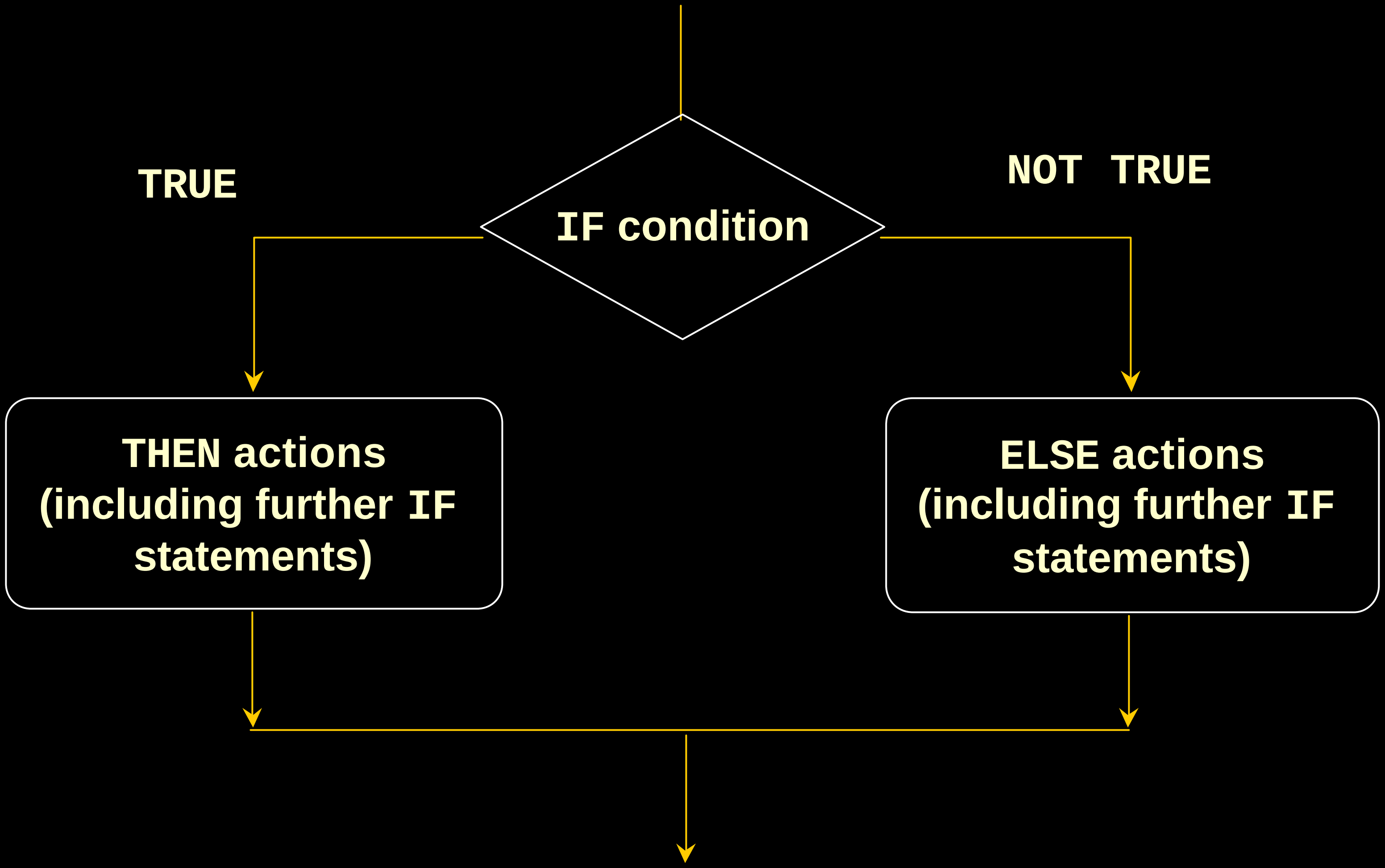
Compound IF Statements

If the last name is Vargas and the salary is more than 6500:

Set department number to 60.

```
. . .
IF v_ename  = 'Vargas' AND salary > 6500 THEN
   v_deptno := 60;
END IF;
. . .
```

IF-THEN-ELSE Statement Execution Flow

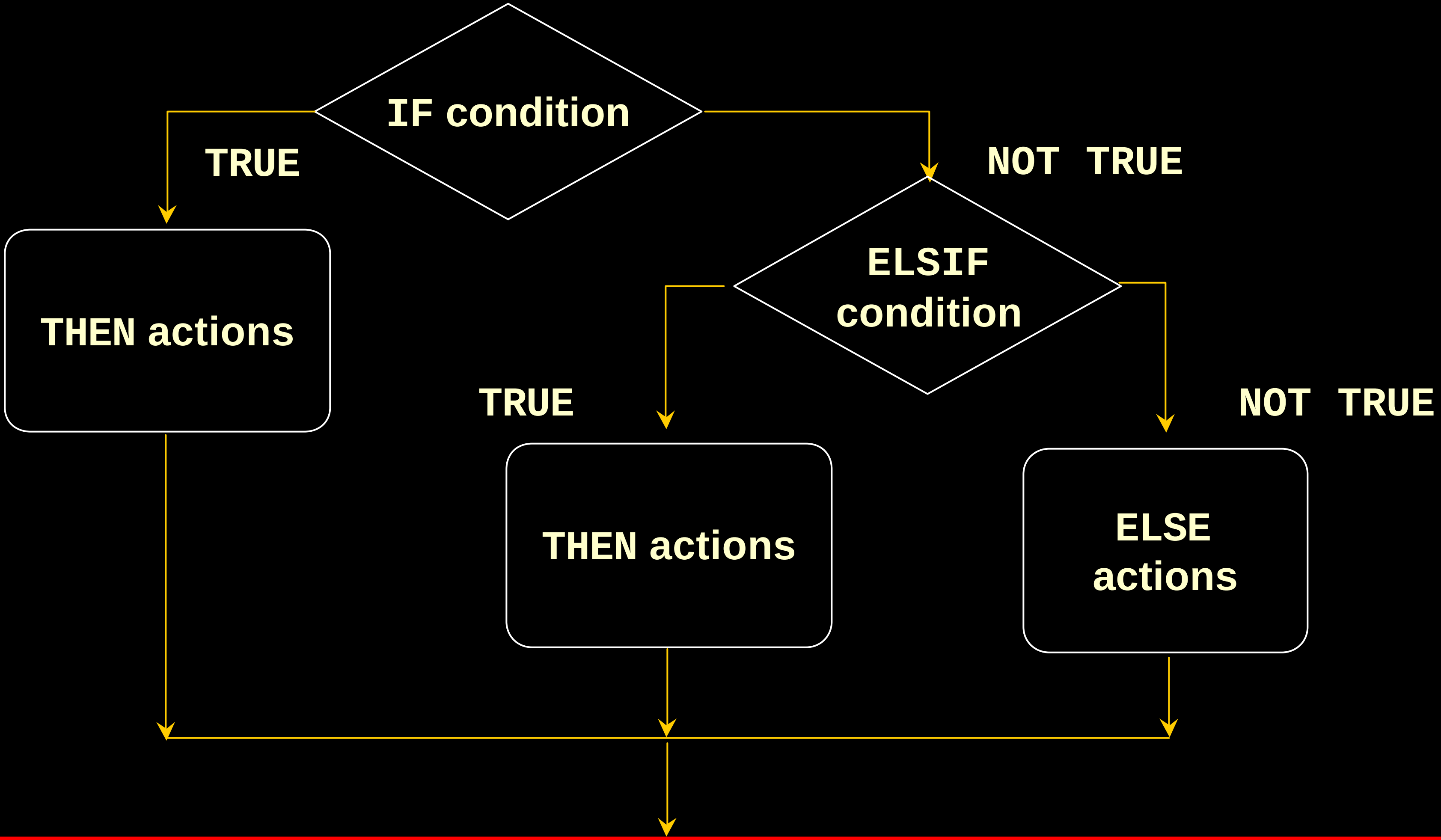


IF-THEN-ELSE Statements

Set a Boolean flag to TRUE if the hire date is greater than five years; otherwise, set the Boolean flag to FALSE.

```
DECLARE
   v_hire_date  DATE := '12-Dec-1990';
   v_five_years BOOLEAN;
BEGIN
. . .
IF MONTHS_BETWEEN(SYSDATE,v_hire_date)/12 > 5 THEN
   v_five_years := TRUE;
ELSE
   v_five_years := FALSE;
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 := 0.2 * v_start;
ELSIF v_start >= 50 THEN
      v_start := 0.5 * v_start;
ELSE
      v_start := 0.1 * v_start;
END IF;
. . .
```

CASE Expressions

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses an expression whose value 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
DECLARE
   v_grade CHAR(1) := UPPER('&p_grade');
   v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal :=
      CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || '
                       Appraisal ' || v_appraisal);
END;
/
```

Handling Nulls

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- In conditional control statements, if the condition yields `NULL`, its associated sequence of statements is not executed.

Boolean Conditions

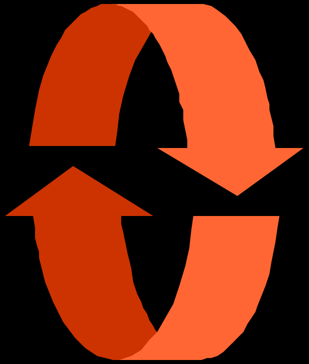
What is the value of V_FLAG in each case?

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

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	?
TRUE	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



ORACLE

Basic Loops

Syntax:

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

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

Basic Loops

Example:

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

WHILE Loops

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 Loops

Example:

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

FOR Loops

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.
- '`lower_bound .. upper_bound`' is required syntax.

FOR Loops

Insert three new locations IDs for the country code of CA and the city of Montreal.

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

FOR Loops

Guidelines

- Reference the counter within the loop only; it is undefined outside the loop.
- Do *not* reference the counter as the target of an assignment.

Guidelines While Using Loops

- Use the basic loop when the statements inside the loop must execute at least once.
- Use the `WHILE` loop if the condition has to be evaluated at the start of each iteration.
- Use a `FOR` loop if the number of iterations is known.

Summary

In this lesson you should have learned to:

Change the logical flow of statements by using control structures.

- Conditional (`IF` statement)
- `CASE` Expressions
- Loops:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop
- `EXIT` statements

5

Working with Composite Data Types

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- Create user-defined PL/SQL records
- Create a record with the `%ROWTYPE` attribute
- Create an `INDEX BY` table
- Create an `INDEX BY` table of records
- Describe the difference between records, tables, and tables of records

PL/SQL Records

- Must contain one or more components of any scalar, `RECORD`, or `INDEX BY` table data type, called fields
- Are similar in structure to records in a third generation language (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]
```

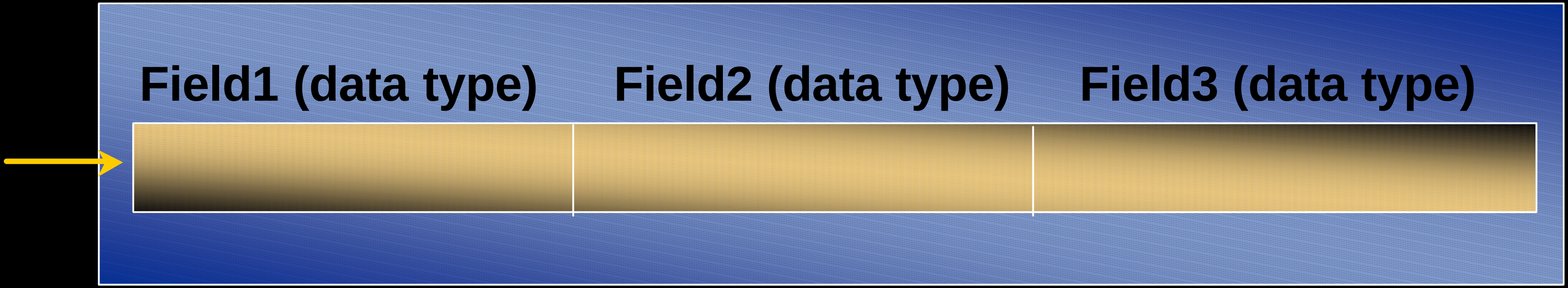
Creating a PL/SQL Record

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

Example:

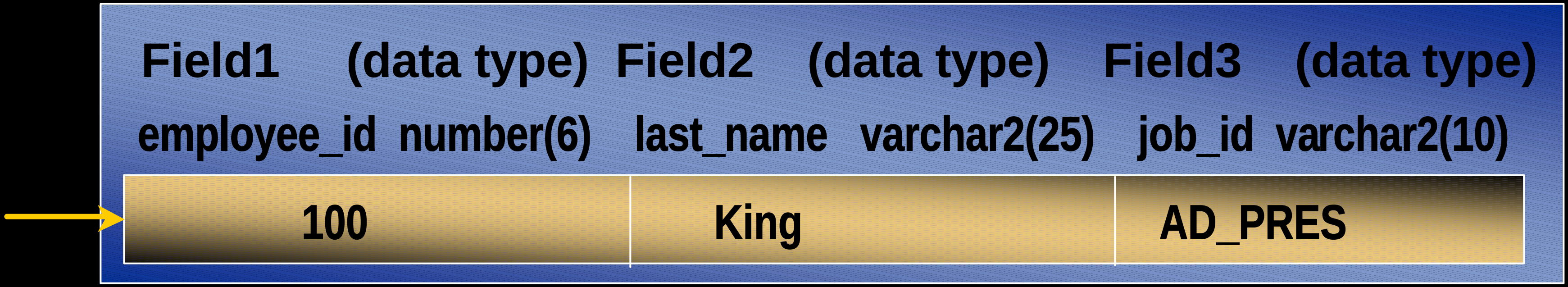
```
...
  TYPE emp_record_type IS RECORD
    (last_name   VARCHAR2(25),
     job_id      VARCHAR2(10),
     salary      NUMBER(8,2));
  emp_record     emp_record_type;
...
```

PL/SQL Record Structure



Field1 (data type) Field2 (data type) Field3 (data type)

Example:



Field1 (data type) Field2 (data type) Field3 (data type)

employee_id number(6) last_name varchar2(25) job_id varchar2(10)

100 King AD_PRES

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 data types from the columns of the table or view.

Advantages of Using %ROWTYPE

- The number and data types of the underlying database columns need not be known.
- The number and data types of the underlying database column may change at run time.
- The attribute is useful when retrieving a row with the SELECT * statement.

The %ROWTYPE Attribute

Examples:

Declare a variable to store the information about a department from the DEPARTMENTS table.

```
dept_record    departments%ROWTYPE;
```

Declare a variable to store the information about an employee from the EMPLOYEES table.

```
emp_record     employees%ROWTYPE;
```

INDEX BY Tables

- Are composed of two components:
  - Primary key of data type BINARY_INTEGER
  - Column of scalar or record data type
- Can increase in size dynamically because they are unconstrained

Creating an INDEX BY Table

Syntax:

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

Declare an INDEX BY table to store names.

Example:

```
...
TYPE ename_table_type IS TABLE OF
                               employees.last_name%TYPE
  INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
...
```

INDEX BY Table Structure

Unique identifier	Column
...	...
1	Jones
2	Smith
3	Maduro
...	...
BINARY_INTEGER	Scalar

Creating an INDEX BY Table

```
DECLARE
  TYPE ename_table_type IS TABLE OF
       employees.last_name%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 INDEX BY Table Methods

The following methods make INDEX BY tables easier to use:

- EXISTS
- COUNT
- FIRST and LAST
- PRIOR
- NEXT
- TRIM
- DELETE

INDEX BY Table of Records

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

Example:

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

Example of INDEX BY Table of Records

```
SET SERVEROUTPUT ON
DECLARE
   TYPE emp_table_type is table of
      employees%ROWTYPE INDEX BY BINARY_INTEGER;
   my_emp_table  emp_table_type;
   v_count       NUMBER(3):= 104;
BEGIN
  FOR i IN 100..v_count
  LOOP
         SELECT * INTO my_emp_table(i) FROM employees
         WHERE employee_id = i;
  END LOOP;
  FOR i IN my_emp_table.FIRST..my_emp_table.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
  END LOOP;
END;
```

Summary

In this lesson, you should have learned to:

- Define and reference PL/SQL variables of composite data types:
  - PL/SQL records
  - INDEX BY tables
  - INDEX BY table of records
- Define a PL/SQL record by using the %ROWTYPE attribute

6

Writing Explicit Cursors

Objectives

After completing this lesson, you should be able to do the following:

- Distinguish between an implicit and an explicit cursor
- Discuss when and why to use an explicit cursor
- Use a PL/SQL record variable
- Write a cursor F0R loop

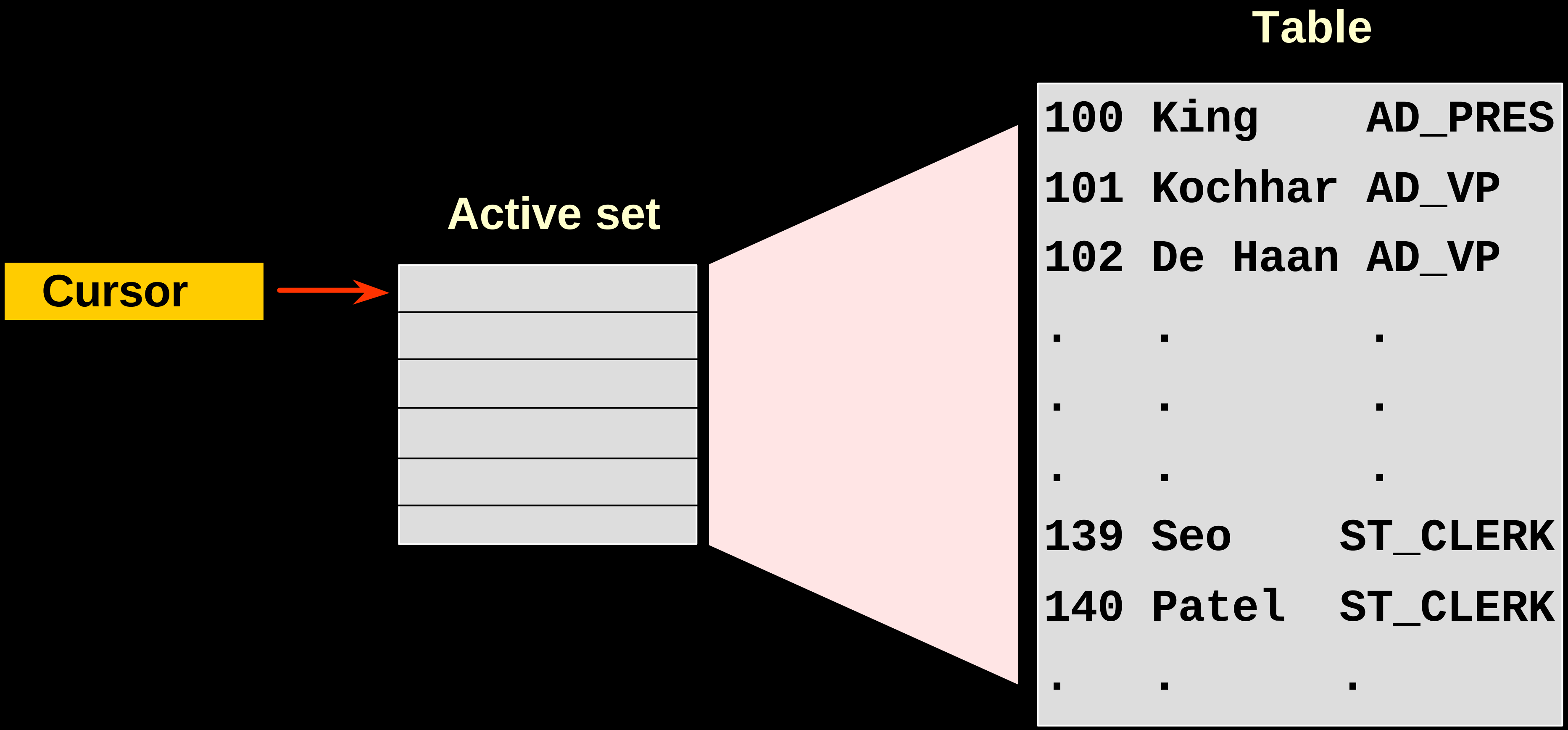
ORACLE

About Cursors

Every SQL statement executed by the Oracle Server has an individual cursor associated with it:

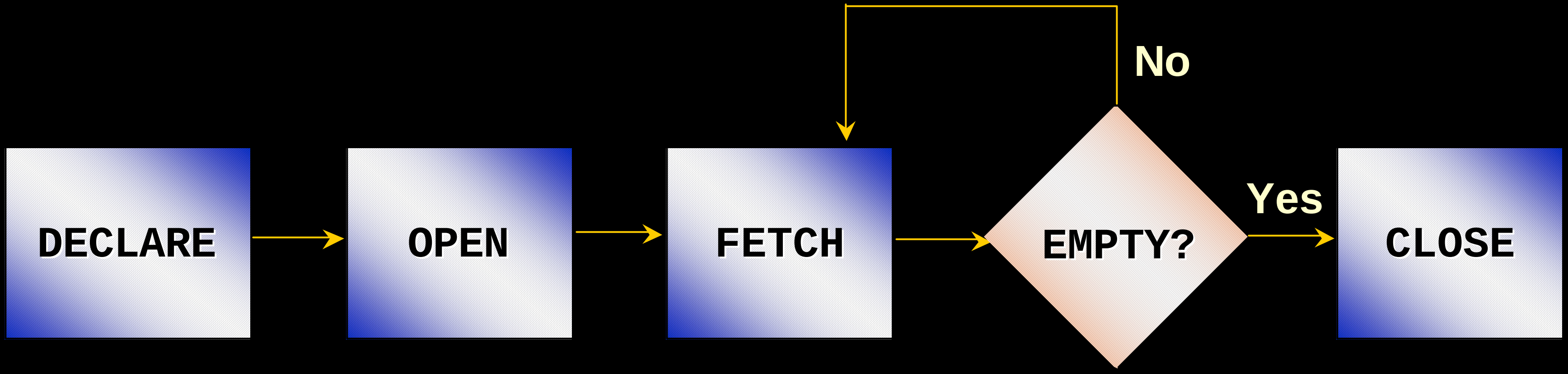
- Implicit cursors: Declared for all DML and PL/SQL SELECT statements
- Explicit cursors: Declared and named by the programmer

Explicit Cursor Functions



```
100 King     AD_PRES
101 Kochhar AD_VP
102 De Haan AD_VP
.   .       .
.   .       .
.   .       .
139 Seo     ST_CLERK
140 Patel   ST_CLERK
.   .       .
```

Controlling Explicit Cursors



- Create a named SQL area

- Identify the active set

- Load the current row into variables

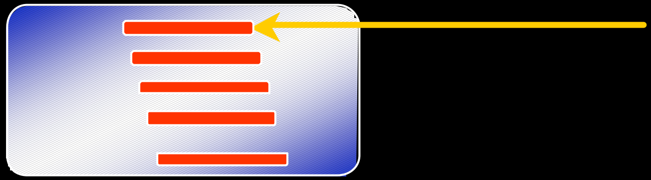
- Test for existing rows
- Return to FETCH if rows are found

- Release the active set

Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor

1. Open the cursor.

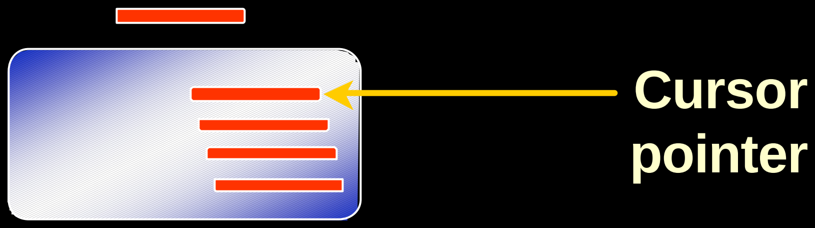


Cursor pointer

Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor

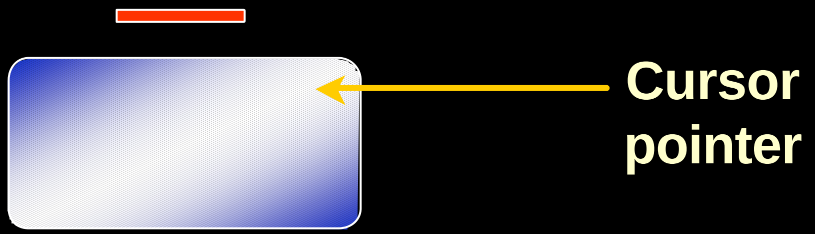
2. Fetch a row using the cursor.



Continue until empty.

Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. 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 employee_id, last_name
    FROM   employees;

  CURSOR dept_cursor IS
    SELECT *
    FROM   departments
    WHERE  location_id = 170;
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 whether the cursor contains rows.

Fetching Data from the Cursor

Example:

```
LOOP
  FETCH emp_cursor INTO v_empno,v_ename;
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  …
END LOOP;
```

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 after 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

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...
```

Controlling Multiple Fetches

- Process several rows from an explicit cursor using a loop.
- Fetch a row with each iteration.
- Use explicit cursor attributes to test the success of each fetch.

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.

Example

```
DECLARE
      v_empno  employees.employee_id%TYPE;
      v_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 v_empno, v_ename;
        EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                          emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)
                              ||'  '|| v_ename);
      END LOOP;
      CLOSE emp_cursor;
END ;
```

Cursors and Records

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

```
DECLARE
  CURSOR emp_cursor IS
    SELECT  employee_id, last_name
    FROM    employees;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
  ...
```

emp_record

employee_id	last_name
100	King

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

Print a list of the employees who work for the sales department.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT last_name, department_id
    FROM   employees;
BEGIN
  FOR emp_record IN emp_cursor LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.department_id = 80 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 last_name, department_id
                     FROM   employees) LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Summary

In this lesson you should have learned to:

- Distinguish cursor types:
  - Implicit cursors: used for all `DML` statements and single-row queries
  - Explicit cursors: used for queries of zero, one, or more rows
- Manipulate explicit cursors
- Evaluate the cursor status by using cursor attributes
- Use cursor `FOR` loops

Practice 6 Overview

This practice covers the following topics:

- Declaring and using explicit cursors to query rows of a table
- Using a cursor FOR loop
- Applying cursor attributes to test the cursor status

7

Advanced Explicit Cursor Concepts

Objectives

After completing this lesson, you should be able to do the following:

- Write a cursor that uses parameters
- Determine when a `FOR UPDATE` clause in a cursor is required
- Determine when to use the `WHERE CURRENT OF` clause
- Write a cursor that uses a subquery

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

Pass the department number and job title to the WHERE clause, in the cursor SELECT statement.

```
DECLARE
  CURSOR emp_cursor
  (p_deptno NUMBER, p_job VARCHAR2) IS
     SELECT employee_id, last_name
     FROM   employees
     WHERE  department_id = p_deptno
     AND    job_id = p_job;
BEGIN
  OPEN emp_cursor (80, 'SA_REP');
  . . .
  CLOSE emp_cursor;
  OPEN emp_cursor (60, 'IT_PROG');
  . . .
END;
```

Cursors with Subqueries

Example:

```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.department_id, t1.department_name,
           t2.staff
    FROM   departments t1, (SELECT department_id,
                                   COUNT(*) AS STAFF
                            FROM employees
                            GROUP BY department_id) t2
    WHERE t1.department_id = t2.department_id
    AND   t2.staff >= 3;
...
```

8

Handling Exceptions

Objectives

After completing this lesson, you should be able to do the following:

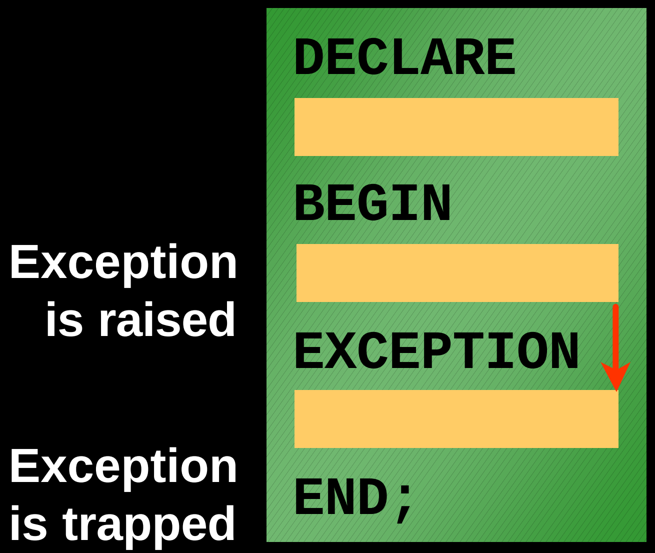
- Define PL/SQL exceptions
- Recognize unhandled exceptions
- List and use different types of PL/SQL exception handlers
- Trap unanticipated errors
- Describe the effect of exception propagation in nested blocks
- Customize PL/SQL exception messages

Handling Exceptions with PL/SQL

- An exception is an 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

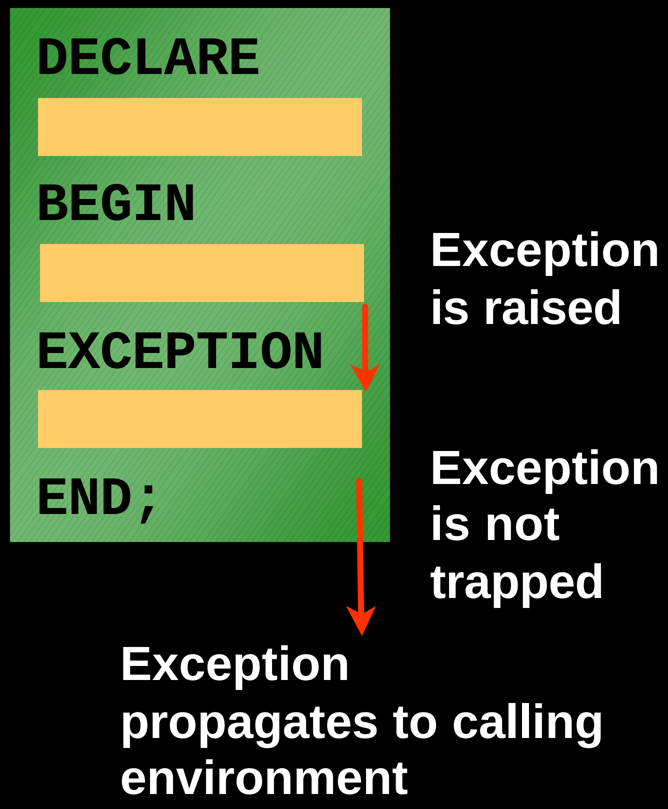


```
DECLARE

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
- Nonpredefined Oracle Server

} Implicitly raised

- User-defined   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

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

Trapping Predefined Oracle Server Errors

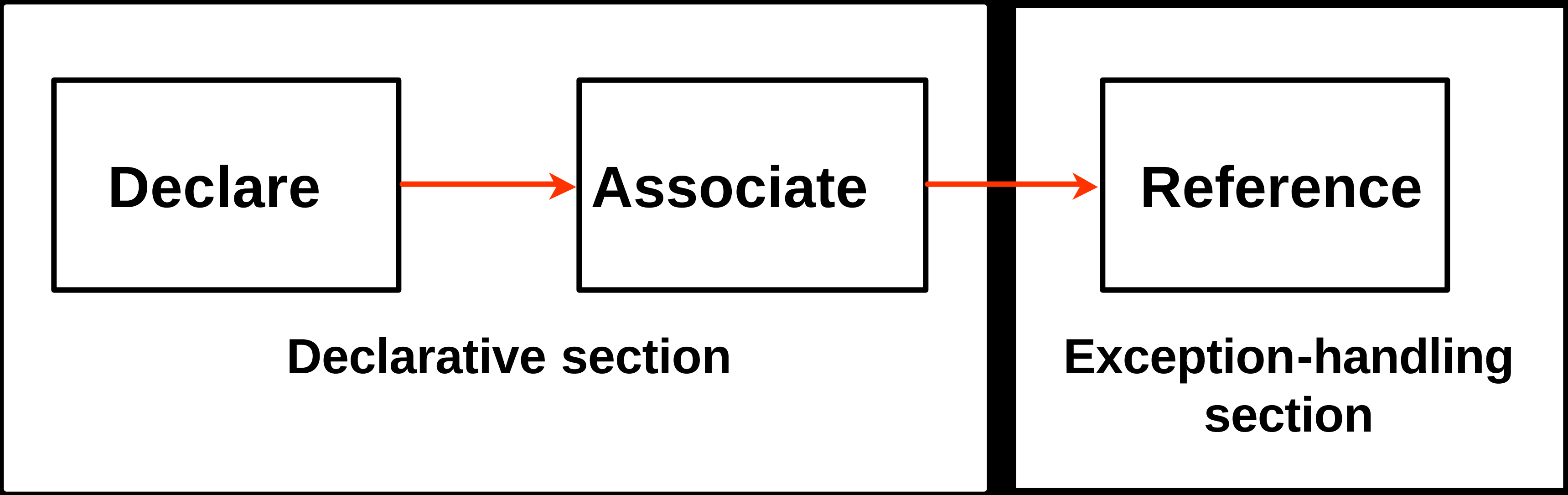
- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

Predefined Exceptions

Syntax:

```
BEGIN
. . .
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;

  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

Trapping Nonpredefined Oracle Server Errors



Declare

Associate

Reference

Declarative section

Exception-handling section

Name the exception

Code the `PRAGMA EXCEPTION_INIT`

Handle the raised exception

Nonpredefined Error

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

```
DEFINE p_deptno = 10
DECLARE
  e_emps_remaining EXCEPTION;                    1
  PRAGMA EXCEPTION_INIT                          2
    (e_emps_remaining, -2292);
BEGIN
  DELETE FROM departments
  WHERE  department_id = &p_deptno;
  COMMIT;
EXCEPTION
  WHEN e_emps_remaining  THEN                    3
   DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
   TO_CHAR(&p_deptno) || '.  Employees exist. ');
END;
```

Functions for Trapping Exceptions

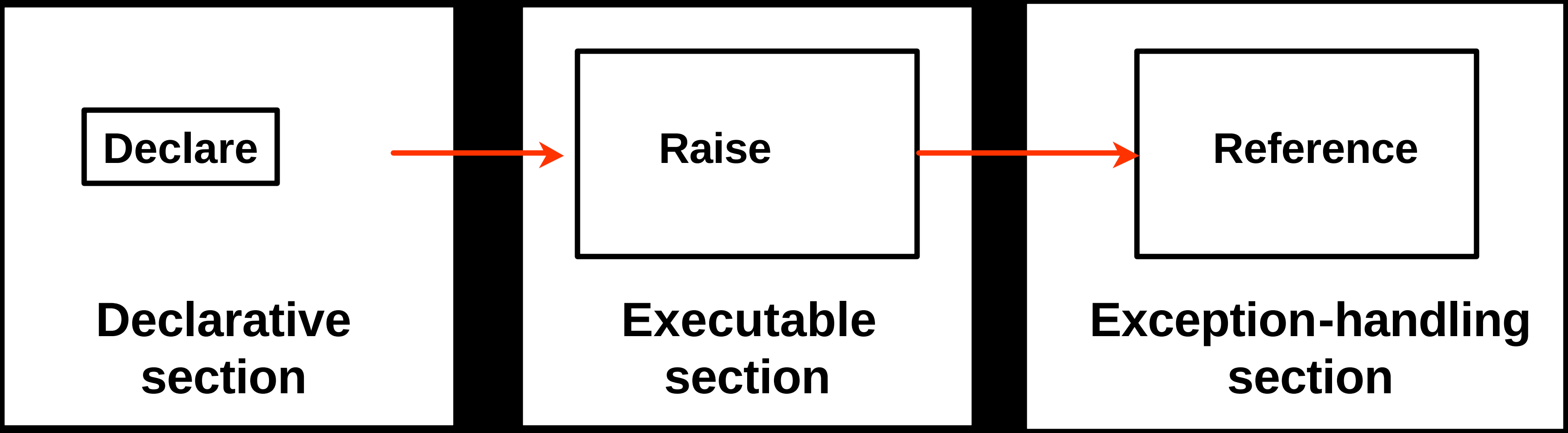
- **`SQLCODE`: Returns the numeric value for the error code**
- **`SQLERRM`: Returns the message associated with the error number**

Functions for Trapping Exceptions

Example:

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

Trapping User-Defined Exceptions



Name the exception.

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

Handle the raised exception.

User-Defined Exceptions

Example:

```
DEFINE p_department_desc = 'Information Technology '
DEFINE P_department_number = 300
```

```
DECLARE
  e_invalid_department EXCEPTION;                    1
BEGIN
  UPDATE      departments
  SET         department_name = '&p_department_desc'
  WHERE       department_id = &p_department_number;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;                      2
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_department THEN                     3
    DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
```

Calling Environments

*i*SQL*Plus	Displays error number and message to screen
Procedure Builder	Displays error number and message to screen
Oracle Developer Forms	Accesses error number and message in a trigger by means of the `ERROR_CODE` and `ERROR_TEXT` packaged functions
Precompiler application	Accesses exception number through the `SQLCA` data structure
An enclosing PL/SQL block	Traps exception in exception-handling routine of enclosing block

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;
    END;
  END LOOP;
EXCEPTION
  WHEN e_integrity THEN ...
  WHEN e_no_rows THEN ...
END;
```

The RAISE_APPLICATION_ERROR Procedure

Syntax:

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

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

The 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

RAISE_APPLICATION_ERROR

Executable section:

```
BEGIN
...
  DELETE FROM employees
     WHERE  manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
     RAISE_APPLICATION_ERROR(-20202,
       'This is not a valid manager');
  END IF;
   ...
```

Exception section:

```
...
EXCEPTION
     WHEN NO_DATA_FOUND THEN
       RAISE_APPLICATION_ERROR (-20201,
         'Manager is not a valid employee.');
END;
```

Summary

In this lesson, you should have learned that:

- Exception types:
  - Predefined Oracle server error
  - Nonpredefined Oracle server error
  - User-defined error
- Exception trapping
- Exception handling:
  - Trap the exception within the PL/SQL block.
  - Propagate the exception.