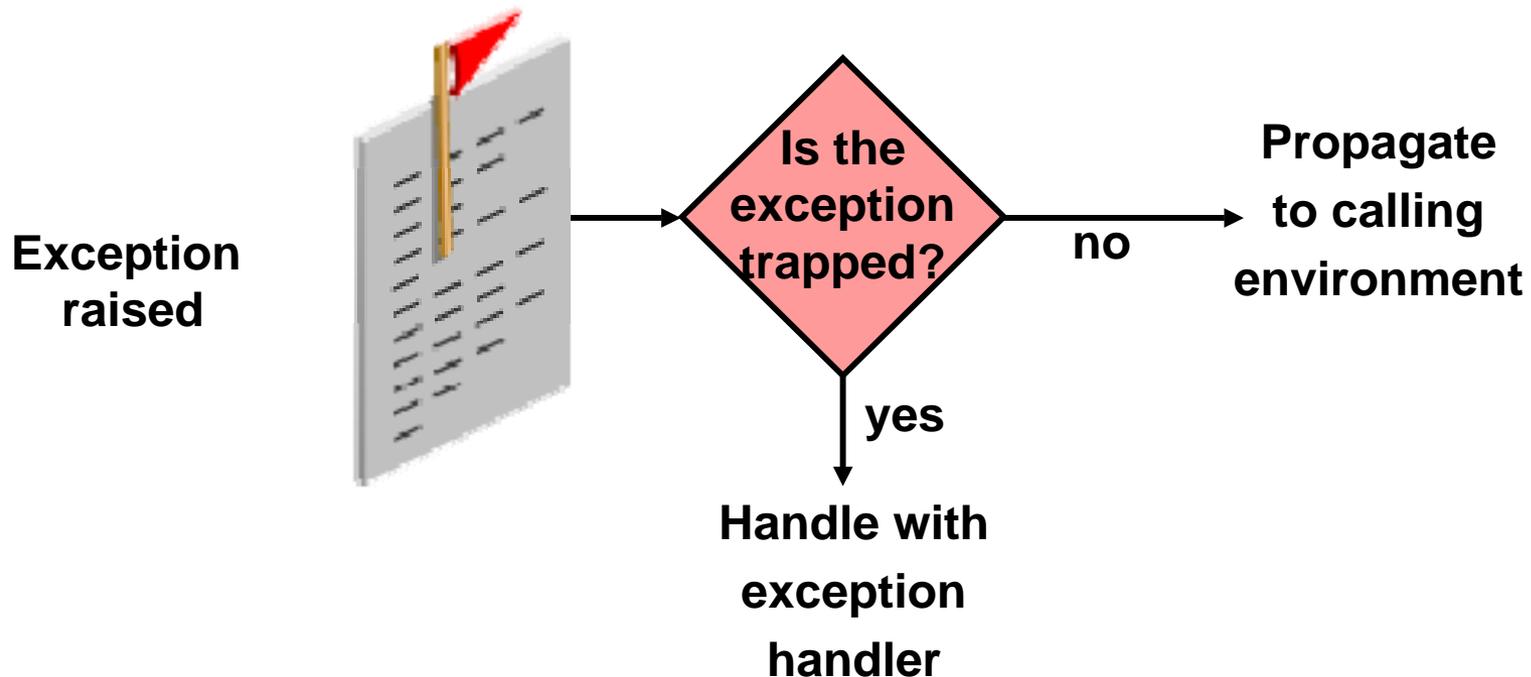


Scope of Exceptions in Nested Blocks

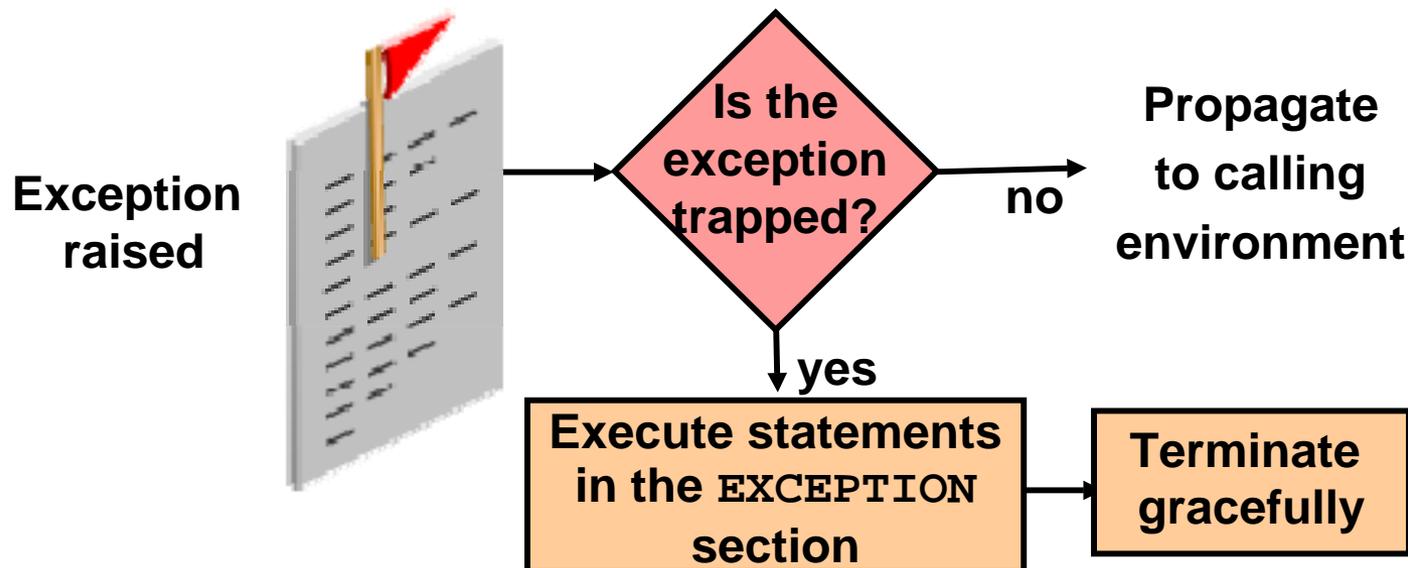
An exception can be dealt with in two ways:

- Handling it (“trapping it”) in the block in which it occurs
- Propagating it to the calling environment



Trapping Exceptions with a Handler

Include an `EXCEPTION` section in your PL/SQL program to trap exceptions. If the exception is raised in the executable section of the block, processing is handled by the corresponding exception handler in the exception section of the same block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or to the calling environment. The PL/SQL block terminates successfully.



Handling Exceptions in an Inner Block

In this example, an error occurs during the execution of the inner block. The inner block's `EXCEPTION` section deals with the exception successfully, and PL/SQL considers that this exception is now finished with. The outer block resumes execution as normal.

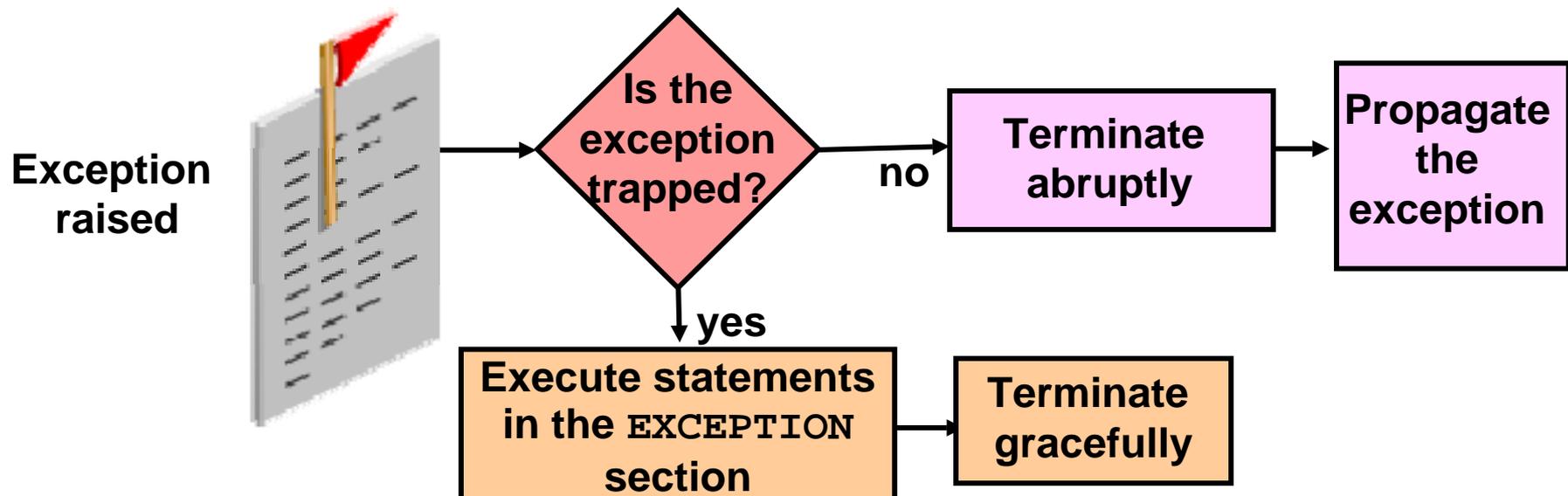
```
BEGIN -- outer block
...

BEGIN -- inner block
... - exception_name occurs here
...
EXCEPTION
  WHEN exception_name THEN -- handled here
  ...
END; -- inner block terminates successfully

... -- outer block continues execution
END;
```

Propagating Exceptions to an Outer Block

If the exception is raised in the executable section of the inner block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is **propagated** to an enclosing block.





Propagating Exceptions to an Outer Block continued

In this example, an error occurs during the execution of the inner block. The inner block's `EXCEPTION` section does not deal with the exception. The inner block terminates unsuccessfully and PL/SQL passes the exception to the outer block. The outer block's `EXCEPTION` section successfully handles the exception.

```
BEGIN -- outer block
...
BEGIN -- inner block
... - exception_name occurs here
...
END; -- Inner block terminates unsuccessfully

... -- Remaining code in outer block's executable
... -- section is skipped
EXCEPTION
  WHEN exception_name THEN - outer block handles the exception
  ...
END;
```



Propagating Exceptions in a Subblock

If a PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates to successive enclosing blocks until it finds a handler.

When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.

If none of these blocks handle the exception, an unhandled exception occurs in the host environment (for example Application Express).

Good Programming Practices



Good Programming Practices

- Conversions
 - Do not rely on implicit data type conversions because they can be slower and the rules may change in later software releases.
- Declaring and initializing PL/SQL variables
 - Use meaningful names
 - Declare one identifier per line for readability and code maintenance.
 - Use the `NOT NULL` constraint when the variable must hold a value.
 - Avoid using column names as identifiers.
 - Use the `%TYPE` attribute to declare a variable according to another previously declared variable or database column.

 **Good Programming Practices continued**

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



Good Programming Practices continued

- Add single line comments: two dashes (--) and multiple line comments: between the symbols “/*” and “*/” to your code

Example:

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

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

 **Good Programming Practices continued**

Use case conventions to help you distinguish keywords from named objects.

Category	Case Convention	Examples
SQL keywords	Uppercase	SELECT, INSERT
PL/SQL keywords	Uppercase	DECLARE, BEGIN, IF
Data types	Uppercase	VARCHAR2, BOOLEAN
Identifiers and parameters	Lowercase	v_sal, emp_cursor, g_sal, p_empno
Database tables and columns	Lowercase	employees, employee_id, department_id

 **Good Programming Practices continued**

Use naming conventions that are clear, consistent and unambiguous. One commonly-used convention is to name:

- variables starting with v_
 - constants starting with c_
 - parameters (passed to procedures and functions) starting with p_
- Indent each level of code for clarity

```
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;
```

Review of SQL DML



Data Manipulation Language (DML)

You can use DML commands to modify the data in a database table.

INSERT

The `INSERT` statement is used to add new rows to a table and contains three components:

- The name of the table
- The names of the columns in the table to populate
- Values for the columns

```
INSERT INTO art (id, title, artist, description)
VALUES (35, 'Mona Lisa', 'Leonardo da Vinci', 'oil painting');
```



INSERT

You may insert values in a table without listing the column names. This is not recommended due to the potential of columns being added to the table.

The values for each column must match the default order in which they appear in the table, and a value must be provided for each column.

```
INSERT INTO art
VALUES (40, 'The Violinist', 'Marc Chagall', 'oil painting');
```



UPDATE

The `UPDATE` statement is used to modify existing data in a table. It requires at least three values:

- The name of the table
- The name of the column in the table to modify
- A corresponding value or subquery for the column
- A condition can be optionally added that identifies the rows to be modified by the statement.

```
UPDATE art
SET description = 'Stained Glass Window'
WHERE id = 40;
```

ID	TITLE	ARTIST	DESCRIPTION
10	Guerrica	Pablo Picasso	oil painting
20	Skriget	Edvard Munch	oil painting
30	Femmes de Tahiti (Sur la plage)	Paul Gauguin	oil painting
35	Mona Lisa	Leonardo da Vinci	oil painting
40	Stained Glass window at the United Nations Building, New York	Leonardo da Vinci	oil painting

Stained



DELETE

The `DELETE` statement is used to remove existing rows from a table. The statement contains:

- The name of the table
- A condition can be optionally added that identifies the rows to be deleted.

In the example shown, the highlighted row will be deleted.

ID	TITLE	ARTIST	DESCRIPTION
10	Guernica	Pablo Picasso	oil painting
20	Skriget	Edvard Munch	oil painting
30	Femmes de Tahiti (Sur la plage)	Paul Gauguin	oil painting
35	Mona Lisa	Leonardo da Vinci	oil painting
40	Stained Glass window at the United Nations Building, New York	Marc Chagall	Stained Glass Window

```
DELETE FROM art
WHERE id = 10;
```


MERGE

The `MERGE` statement will `INSERT` and/or `UPDATE` a target table, based on matching values in a source table. If a matching value is missing, a new row is inserted. If a matching value exists, but needs to be changed, then `MERGE` will update it.

Merge the data from the ITEMS table into:

ITEM_ID	TITLE	ARTIST	DESCRIPTION
1	Madonna	Edward Munch	oil painting
3	Femmes de Tahiti (Sur la plage)	Paul Gauguin	oil painting
4	Stained Glass window at the United Nations Building, New York	Paul Gauguin	oil painting
35	Mona Lisa	Leonardo da Vinci	Small Oil Painting on Wood

The ART table

ITEM_ID	TITLE	ARTIST	DESCRIPTION
20	Skriqet	Edward Munch	oil painting
30	Femmes de Tahiti (Sur la plage)	Paul Gauguin	oil painting
35	Mona Lisa	Leonardo da Vinci	oil painting
40	Stained Glass window at the United Nations Building, New York	Marc Chagall	Stained Glass Window



MERGE

The statement shown uses ITEMS (alias *i*) as the source table to merge and update information into ART (alias *a*), the target table.

```
MERGE INTO art a
  USING items i
    ON (a.id = i.item_id)
  WHEN MATCHED
  THEN UPDATE SET
    a.artist      = i.artist,
    a.description = e.description
  WHEN NOT MATCHED
  THEN INSERT
    VALUES(i.item_id, i.title, i.artist, i.description);
```

The ART table
after the MERGE

ID	TITLE	ARTIST	DESCRIPTION
1	Madonna	Edvard Munch	oil painting
3	Femmes de Tahiti (Sur la plage)	Paul Gauguin	oil painting
4	Stained Glass window at the United Nations Building, New York	Marc Chagall	Stained Glass Window
20	Skriqet	Edvard Munch	oil painting
30	Femmes de Tahiti (Sur la plage)	Paul Gauguin	oil painting
35	Mona Lisa	Leonardo de Vinci	Small oil painting on Wood
40	Stained Glass window at the United Nations Building, New York	Marc Chagall	Stained Glass Window

Retrieving Data in PL/SQL



SQL Statements in PL/SQL

You can use the following kinds of SQL statements in PL/SQL:

- `SELECT` to retrieve data from the database
- DML statements such as `INSERT`, `UPDATE`, and `DELETE` to make changes to rows in the database
- Transaction control statements such as `COMMIT`, `ROLLBACK`, or `SAVEPOINT`. You use transaction control statements to make the changes to the database permanent or to discard them.



SQL Statements in PL/SQL

DDL (`CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`) and DCL (`GRANT`, `REVOKE`) cannot be used directly in PL/SQL.

These statements cannot be directly executed because they are constructed and executed at run time. That is, they are dynamic. Static SQL statements are statements that are fixed at the time a program is compiled.



SELECT Statements in PL/SQL

Retrieve data from the database with a `SELECT` statement. The `INTO` clause is mandatory and occurs between the `SELECT` and `FROM` clauses. It is used to specify the names of PL/SQL variables that hold the values that SQL returns from the `SELECT` clause. You must specify one variable for each item selected, and the order of the variables must correspond with the items selected.

```
DECLARE
    v_fname employees.first_name%TYPE;
BEGIN
    SELECT first_name INTO v_fname
        FROM employees WHERE employee_id=200;
    DBMS_OUTPUT.PUT_LINE('First Name is ' || v_fname);
END;
```

 **SELECT Statements in PL/SQL continued**

Retrieve `hire_date` and `salary` for the specified employee.

```
DECLARE
  v_emp_hiredate    employees.hire_date%TYPE;
  v_emp_salary      employees.salary%TYPE;
BEGIN
  SELECT      hire_date, salary
  INTO        v_emp_hiredate, v_emp_salary
  FROM        employees
  WHERE       employee_id = 100;
  DBMS_OUTPUT.PUT_LINE('Hiredate is: ' || v_emp_hiredate
                        || ' and Salary is: '
                        || v_emp_salary);
END;
```

 **SELECT Statements in PL/SQL continued**

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: queries must return exactly one row. A query that returns more than one row or no rows generates an error.

```
DECLARE
    v_salary employees.salary%TYPE;
BEGIN
    SELECT salary INTO v_salary
        FROM employees;
    DBMS_OUTPUT.PUT_LINE(' Salary is : ' || v_salary);
END;
```

```
ORA-01422: exact fetch returns more than requested number of rows
```



Guidelines for Retrieving Data in PL/SQL

- Terminate each SQL statement with a semicolon (;).
- Every value retrieved must be stored in a variable using the `INTO` clause.
- The `WHERE` clause is optional and can contain input variables, constants, literals, or PL/SQL expressions. However, you should fetch only one row and the usage of the `WHERE` clause is therefore needed in nearly all cases.
- Specify the same number of variables in the `INTO` clause as database columns in the `SELECT` clause. Be sure that they correspond positionally and that their data types are compatible.
- Declare the receiving variables using `%TYPE`.



Guidelines for Naming Conventions

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables. This example raises an unhandled run-time exception because in the WHERE clause, the PL/SQL variable name is the same as that of the database column name in the employees table.

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

ORA-01422: exact fetch returns more than requested number of rows

 **Guidelines for Naming Conventions continued**

What is deleted in the following PL/SQL block?

```
DECLARE
  last_name VARCHAR2(25) := 'King';
BEGIN
  DELETE FROM emp_dup WHERE last_name = last_name;
END;
```



Guidelines for Naming Conventions continued

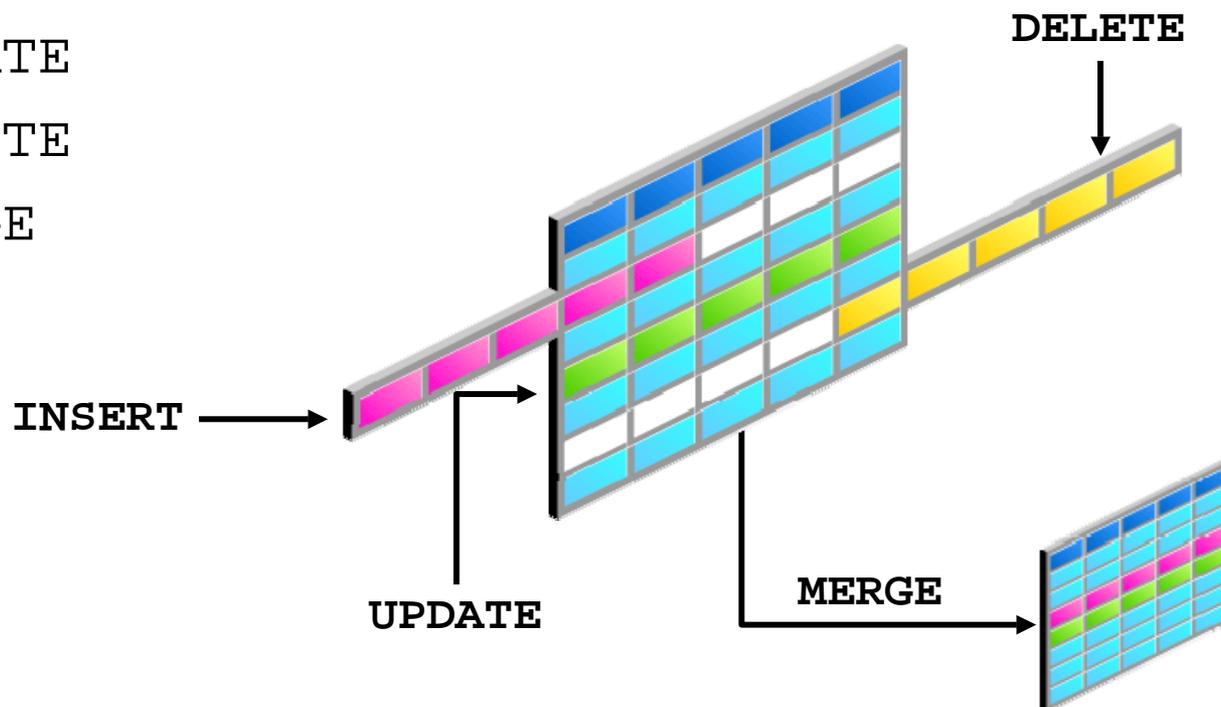
- Use a naming convention to avoid ambiguity in the `WHERE` clause.
- Avoid using database column names as identifiers.
- Errors can arise because PL/SQL checks the database first for a column in the table.
- The names of local variables and formal parameters take precedence over the names of database *tables*.
- The names of database table *columns* take precedence over the names of local variables.

Manipulating Data in PL/SQL

Manipulating Data Using PL/SQL

Make changes to data by using DML commands within your PLSQL block:

- INSERT
- UPDATE
- DELETE
- MERGE





Manipulating Data Using PL/SQL (continued)

- You can issue the DML commands—`INSERT`, `UPDATE`, `DELETE` and `MERGE`—without restriction in PL/SQL. Row locks (and table locks) are released by including `COMMIT` or `ROLLBACK` statements in the PL/SQL code.
 - The `INSERT` statement adds new rows to the table.
 - The `UPDATE` statement modifies existing data in the table.
 - The `DELETE` statement removes rows from the table.
 - The `MERGE` statement selects rows from one table to update and/or insert into another table. The decision whether to update or insert into the target table is based on a condition in the `ON` clause.
- **Note:** `MERGE` is a deterministic statement—that is, you cannot update the same row of the target table multiple times in the same `MERGE` statement. You must have `INSERT` and `UPDATE` object privileges on the target table and `SELECT` privilege on the source table.



Inserting Data

```
BEGIN
  INSERT INTO copy_emp
    (employee_id, first_name, last_name, email,
     hire_date, job_id, salary)
  VALUES(99, 'Ruth', 'Cores',
          'RCORES',SYSDATE, 'AD_ASST', 4000);
END;
```

Updating Data

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



Deleting Data

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

Merging Rows

```
BEGIN
    MERGE INTO copy_emp c
        USING employees e
        ON (e.employee_id = c.employee_id)
    WHEN MATCHED THEN
        UPDATE SET
            c.first_name      = e.first_name,
            ...
    WHEN NOT MATCHED THEN
        INSERT VALUES(e.employee_id, e.first_name, ...e.department_id);
END;
```



Getting information from a Cursor

Look again at the `DELETE` statement in this PL/SQL block.

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

It would be useful to know how many `COPY_EMP` rows were deleted by this statement.

To obtain this information, we need to understand cursors.



What is a Cursor?

Every time a SQL statement is about to be executed, the Oracle server allocates a private memory area called an implicit cursor to store the SQL statements and the data which it uses.

Because this memory area is automatically managed by the Oracle server, you have no direct control over it. However, you can use predefined PL/SQL variables, called implicit cursor attributes, to find out how many rows were processed by the SQL statement.

An implicit cursor is always automatically named “SQL”

Cursor Attributes for Implicit Cursors

Cursor attributes are automatically declared variables which allow you to evaluate what happened when a cursor was last used. Attributes for implicit cursors are prefaced with “SQL”. Use these attributes in PL/SQL statements, but not in SQL statements. Using 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 the number of rows affected by the most recent SQL statement

 **Example using Cursor attributes:**

```
BEGIN
  UPDATE      copy_emp
    SET       salary = salary + 100
    WHERE     job_id = 'ST_CLERK';
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||
                        ' rows updated.');
```

END;

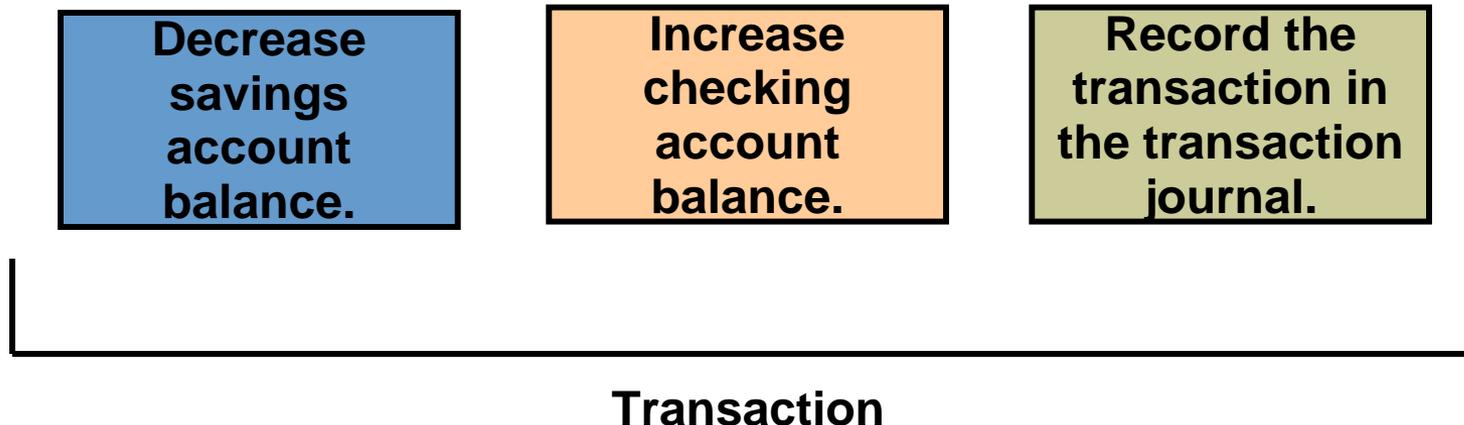
Using Transaction Control Statements



Database Transaction

A transaction is an inseparable list of database operations, which must be executed either in its entirety or not at all. Transactions maintain data integrity and guarantee that the database will always be in a consistent state.

To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:



 **Example of a Transaction continued**

What would happen if there were insufficient funds in the savings account? Would the funds still be added to the checking account? Would an entry be logged in the transaction journal? What do you think *should* happen?

**Decrease
savings
account.**

```
UPDATE savings_accounts  
SET balance = balance - 500  
WHERE account = 3209;
```

**Increase
checking
account.**

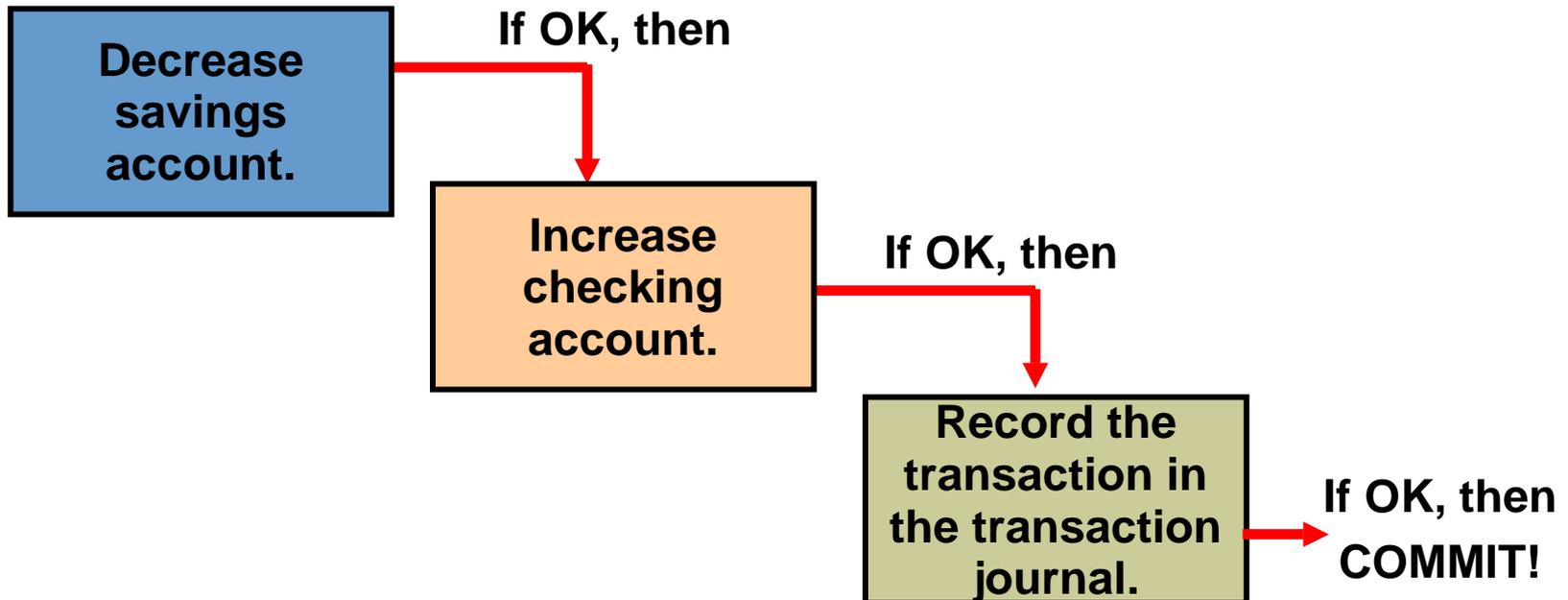
```
UPDATE checking_accounts  
SET balance = balance + 500  
WHERE account = 3208;
```

**Record the
transaction in
the transaction
journal.**

```
INSERT INTO journal VALUES  
(journal_seq.NEXTVAL, '1B'  
3209, 3208, 500);
```

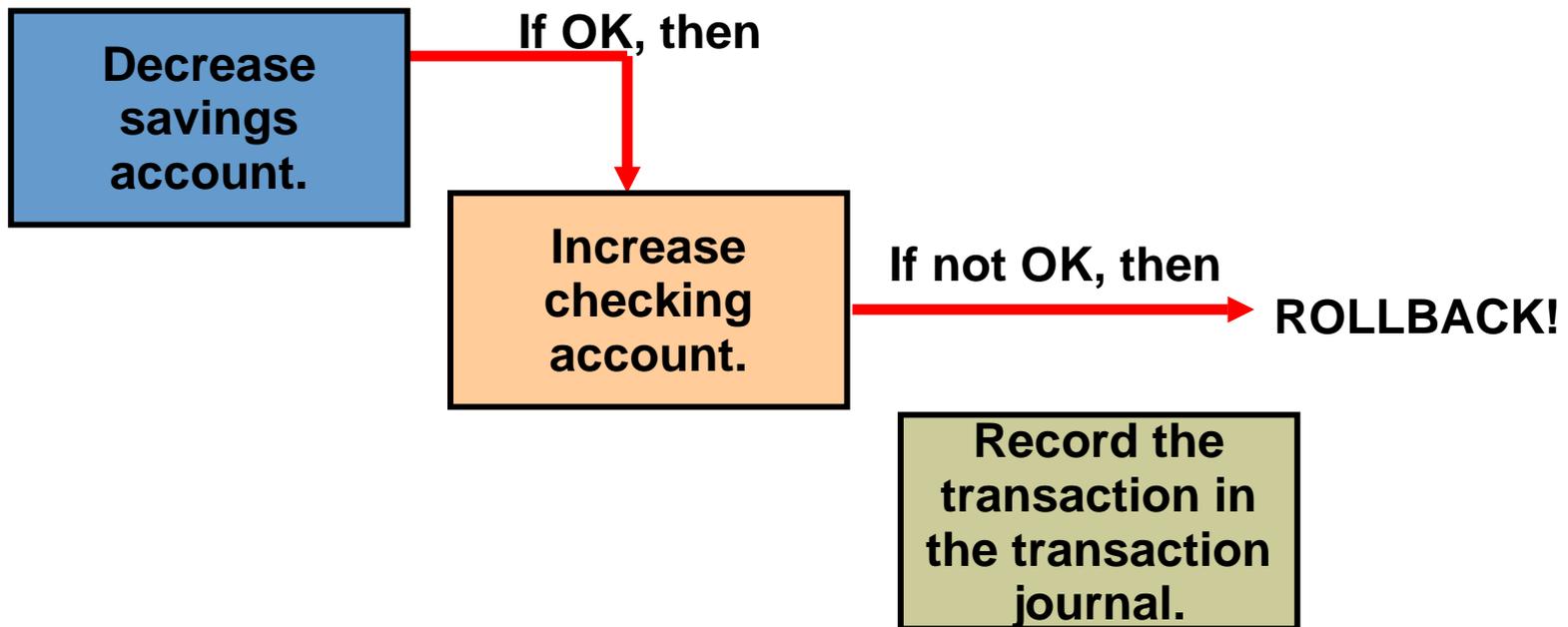
Example of a Transaction (continued)

If all three SQL statements can be performed to maintain the accounts in proper balance, the effects of the transaction can be committed, or applied to the database tables.



Example of a Transaction

However, if a problem such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, the entire transaction must be rolled back (ie reversed out) so that the balance of all accounts is correct.



Transaction Control Statements

You use transaction control statements to make the changes to the database permanent or to discard them. The three main transaction control statements are:

- COMMIT
- ROLLBACK
- SAVEPOINT

The transaction control commands are valid in PL/SQL and therefore can be directly used in the executable or exception section of a PL/SQL block.



COMMIT is used to make the database changes permanent. If a transaction ends with a `COMMIT` statement, all the changes made to the database during that transaction are made permanent.

```
BEGIN
  INSERT INTO pairtable VALUES (1, 2);
  COMMIT;
END;
```

Important: The keyword `END` signals the end of a PL/SQL block, not the end of a transaction.

`ROLLBACK` is for discarding any changes that were made to the database after the last `COMMIT`. If the transaction fails, or ends with a `ROLLBACK`, then none of the statements takes effect.

```
BEGIN
  INSERT INTO pairtable VALUES (3, 4);
  ROLLBACK;
  INSERT INTO pairtable VALUES (5, 6);
  COMMIT;
END;
```

 **SAVEPOINT is used to mark an intermediate point in transaction processing.**

```
BEGIN
  INSERT INTO pairtable VALUES (7, 8);
  SAVEPOINT my_sp_1;
  INSERT INTO pairtable VALUES (9, 10);
  SAVEPOINT my_sp_2;
  INSERT INTO pairtable VALUES (11, 12);
  ROLLBACK to my_sp_1;
  INSERT INTO pairtable VALUES (13, 14);
  COMMIT;
END;
```

Only ROLLBACK can be used to a SAVEPOINT.

Conditional Control: IF Statements

Controlling the Flow of Execution

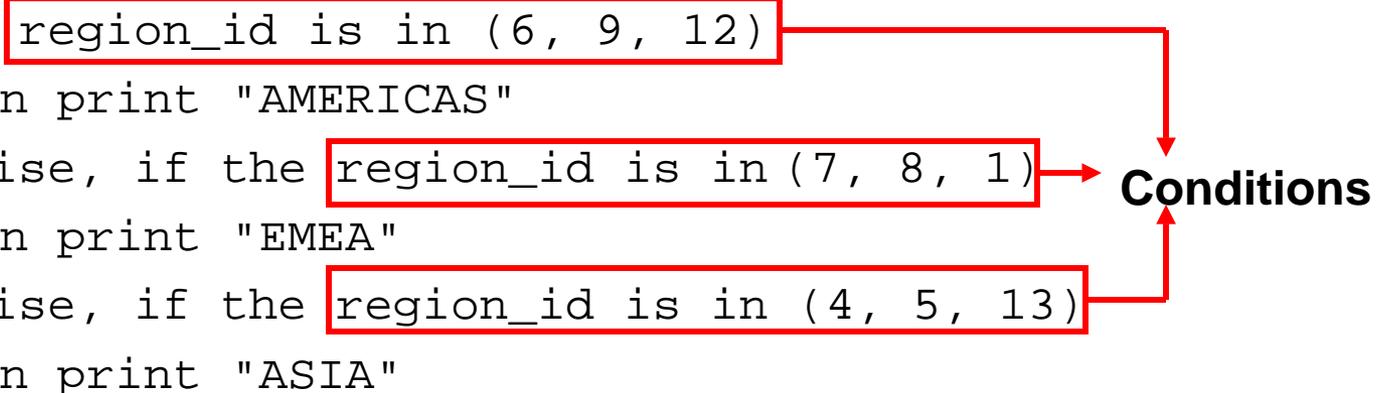
You can change the logical flow of statements within the PL/SQL block with any of three control structures.

IF statement

The `IF` statement contains alternative courses of action in a block based on conditions. A condition is an expression with a `TRUE` or `FALSE` value that is used to make a decision.

Consider the following example:

```
if the region_id is in (6, 9, 12)
  then print "AMERICAS"
otherwise, if the region_id is in (7, 8, 1)
  then print "EMEA"
otherwise, if the region_id is in (4, 5, 13)
  then print "ASIA"
```



Conditions



CASE Statements

CASE statements are similar to IF statements in that they also determine a course of action based on conditions. They are different in that they can be used outside of a PL/SQL block in a SQL statement.

LOOP control structures

LOOP control structures are repetition statements that enable you to execute statements in a PL/SQL block repeatedly. There are three types of loop control structures supported by PL/SQL, **BASIC**, **FOR**, and **WHILE**.

CASE statements and loops will be discussed in later lessons.



IF Statements

The structure of the PL/SQL `IF` statement is similar to the structure of `IF` statements in other procedural languages. It enables PL/SQL to perform actions selectively based on conditions.

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

- *condition* Is a Boolean variable or expression that returns TRUE, FALSE, or NULL
- THEN Introduces a clause that associates the Boolean expression with the sequence of statements that follows it
- *statements* Can be one or more PL/SQL or SQL statements. (They may include further IF statements containing several nested IF, ELSE, and ELSIF statements.) The statements in the THEN clause are executed only if the condition in the associated IF clause evaluates to TRUE.



IF Statements continued

- **ELSIF** Is a keyword that introduces a Boolean expression. (If the first condition yields FALSE or NULL, then the ELSIF keyword introduces additional conditions.)
- **ELSE** Introduces the default clause that is executed if and only if none of the earlier predicates (introduced by IF and ELSIF) is TRUE. The tests are executed in sequence so that a later predicate that might be true is pre-empted by an earlier predicate that is true.
- **END IF;** Marks the end of an IF statement

Note: **ELSIF** and **ELSE** are optional in an **IF** statement. You can have any number of **ELSIF** keywords but only one **ELSE** keyword in your **IF** statement.

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



Examples:

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

```
DECLARE
  v_myage NUMBER:=31;
BEGIN
  IF v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
```



Examples continued:

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

```
DECLARE
  v_myage          NUMBER          := 31;
  v_myfirstname    VARCHAR2(11)    := 'Christopher';
BEGIN
  IF v_myfirstname = 'Christopher' AND v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child named Christopher');
  END IF;
END;
```

 **Examples continued:**

```
DECLARE
  v_myage NUMBER;
BEGIN
  IF v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
```

In this example, the `myage` variable is declared but is not initialized. The condition in the `IF` statement returns `NULL`, and not `TRUE` or `FALSE`. In such a case, the control goes to the `ELSE` statement because just like `FALSE`, `NULL` is not `TRUE`.



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 a condition yields `NULL`, it behaves just like a `FALSE`, and the associated sequence of statements is not executed.
- Consider the following example:

```
x := 5;  
y := NULL;
```

```
...  
IF x != y THEN -- yields NULL, not TRUE  
  -- sequence_of_statements that are not executed  
END IF;
```

- You may expect the sequence of statements to execute because `x` and `y` seem unequal. But, nulls are indeterminate. Whether or not `x` is equal to `y` is unknown. Therefore, the `IF` condition yields `NULL` and the sequence of statements is bypassed.



Guidelines for Using `IF` Statements

- You can perform actions selectively when a specific condition is being met.
- When writing code, remember the spelling of the keywords:
 - `ELSIF` is one word.
 - `END IF` is two words.
- If the controlling Boolean condition is `TRUE`, then the associated sequence of statements is executed; if the controlling Boolean condition is `FALSE` or `NULL`, then the associated sequence of statements is passed over. Any number of `ELSIF` clauses is permitted.
- Indent the conditionally executed statements for clarity.

Conditional Control: CASE Statements



CASE Expressions

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

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

 **Example:**

```
DECLARE
    v_grade      CHAR(1) := 'A';
    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;
```

```
Grade:  A
Appraisal Excellent

Statement processed.
```



Examples continued

PL/SQL also provides a searched CASE expression. A searched CASE expression has no selector. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type.

```
DECLARE
    v_grade      CHAR(1) := 'A';
    v_appraisal  VARCHAR2(20);
BEGIN
    v_appraisal :=
        CASE
            WHEN v_grade = 'A' AND v_appraisal IS NULL
                THEN 'Excellent'
            WHEN v_grade IN ('B','C') THEN 'Good'
            ELSE 'No such grade'
        END;
    DBMS_OUTPUT.PUT_LINE ('Grade: ' || v_grade ||
                          ' Appraisal ' || v_appraisal);
END;
```

 Examples continued

CASE Statement

```
DECLARE
  v_deptid    departments.department_id%TYPE;
  v_deptname  departments.department_name%TYPE;
  v_emps      NUMBER;
  v_mngid     departments.manager_id%TYPE := 108;
BEGIN
  CASE v_mngid
    WHEN 108 THEN
      SELECT department_id, department_name
         INTO v_deptid, v_deptname FROM departments
        WHERE manager_id=108;
      SELECT count(*) INTO v_emps FROM employees
        WHERE department_id=v_deptid;
    WHEN 200 THEN
      ...
  END CASE;
  DBMS_OUTPUT.PUT_LINE ('You are working in the ' || v_deptname |
  ' department. There are ' || v_emps || ' employees in this
  department');
END;
```



How Is the `CASE` Expression Different from the `CASE` Statement?

- The `CASE` expression evaluates the condition and returns a value.
- The `CASE` statement evaluates the condition and performs an action.
- A `CASE` statement can be a complete PL/SQL block.
- `CASE` statements end with `END CASE ;`.
- `CASE` expressions end with `END ;`.



Logic Tables

When using IF and CASE statements we often need to combine conditions using AND, OR and NOT. The following Logic Tables show the results of all possible combinations of two conditions.

AND	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>	OR	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>	NOT	
<i>TRUE</i>	TRUE	FALSE	NULL	<i>TRUE</i>	TRUE	TRUE	TRUE	<i>TRUE</i>	FALSE
<i>FALSE</i>	FALSE	FALSE	FALSE	<i>FALSE</i>	TRUE	FALSE	NULL	<i>FALSE</i>	TRUE
<i>NULL</i>	NULL	FALSE	NULL	<i>NULL</i>	TRUE	NULL	NULL	<i>NULL</i>	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	FALSE	?
NULL	TRUE	?
NULL	FALSE	?



Iterative Control: Basic Loops

LOOP Statements

Loops repeat a statement or a sequence of statements multiple times.

PL/SQL provides the following types of loops:

- Basic loops that perform repetitive actions without overall conditions (Covered in this lesson.)
- FOR loops that perform iterative actions based on a counter
- WHILE loops that perform repetitive actions based on a condition





Basic Loops

The simplest form of a `LOOP` statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords `LOOP` and `END LOOP`. Use the basic loop when the statements inside the loop must execute at least once.

Each time the flow of execution reaches the `END LOOP` statement, control is returned to the corresponding `LOOP` statement above it.

Without an `EXIT` statement, the loop would be infinite.

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



Example:

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



The EXIT Statement

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('The square of '
      ||v_counter||' is: '|| POWER(v_counter,2));
    v_counter := v_counter + 1;
    IF v_counter > 10 THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

You can use the `EXIT` statement to terminate a loop. The control passes to the next statement after the `END LOOP` statement. You can issue `EXIT` either as an action within an `IF` statement or as a stand-alone statement within the loop.

 **The EXIT Statement continued**

The EXIT statement must be placed inside a loop.

- If the EXIT condition is placed at the top of the loop (before any of the other executable statements) and that condition is initially true, then the loop will exit and the statements will never execute.
- A basic loop can contain multiple EXIT statements, but it is recommended to have only one EXIT point.



The EXIT WHEN Statement

You can attach a `WHEN` clause to allow conditional termination of the loop. When the `EXIT` statement is encountered, the condition in the `WHEN` clause is evaluated. If the condition yields `TRUE`, then the loop ends and control passes to the next statement after the loop.

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('The square of '
      || v_counter || ' is: ' || POWER(v_counter,2));
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

Iterative Control: WHILE and FOR Loops

 **WHILE Loops**

You can use the `WHILE` loop to repeat a sequence of statements until the controlling condition is no longer `TRUE`. The condition is evaluated at the start of each iteration. The loop terminates when the condition is `FALSE` or `NULL`. If the condition is `FALSE` or `NULL` at the start of the loop, then no further iterations are performed.

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

- In the syntax:
 - *condition* Is a Boolean variable or expression (`TRUE`, `FALSE`, or `NULL`)
 - *statement* Can be one or more PL/SQL or SQL statements

 **WHILE Loops continued**

If the variables involved in the conditions do not change, then the condition remains TRUE and the loop does not terminate. If the condition yields NULL, then the loop is bypassed and the control passes to the next statement.

Example:

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

 **FOR Loops**

FOR loops have the same general structure as the basic loop. In addition, they have a control statement before the LOOP keyword to set the number of iterations that PL/SQL performs.

```
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 the required syntax.



FOR Loops continued

- In the syntax:
 - *counter* Is an implicitly declared integer whose value automatically increases or decreases (decreases if the `REVERSE` keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached
 - `REVERSE` Causes the counter to decrement with each iteration from the upper bound to the lower bound (Note that the lower bound is still referenced first.)
 - *lower_bound* Specifies the lower bound for the range of counter values
 - *upper_bound* Specifies the upper bound for the range of counter values
- Do not declare the counter; it is declared implicitly as an integer.

 **FOR Loops (continued)**

- **Note:** The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower bound and upper bound of the loop range can be literals, variables, or expressions, but must evaluate to integers. The bounds are rounded to integers—that is, $1\frac{1}{3}$ or $\frac{8}{5}$ are valid upper or lower bounds. The lower bound and upper bound are inclusive in the loop range. If the lower bound of the loop range evaluates to a larger integer than the upper bound, then the sequence of statements will not be executed.

For example, the following statement is executed only once:

```
FOR i in 3..3
LOOP
    statement1;
END LOOP;
```

 **FOR Loops (continued)**

You have already learned how to insert three new locations for the country code CA and the city Montreal. The example in this slide shows you the same results as shown in the LOOP and the WHILE loop examples.

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

 **FOR Loop 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.
- Neither loop bound should be `NULL`.
- When writing a `FOR` loop, the lower and upper bounds of a `LOOP` statement do not need to be numeric literals. They can be expressions that convert to numeric values.

Example:

```
DECLARE
  v_lower  NUMBER := 1;
  v_upper  NUMBER := 100;
BEGIN
  FOR i IN v_lower..v_upper LOOP
    ...
  END LOOP;
END;
```



Guidelines For 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.

Iterative Control: Nested Loops



Nested Loops

In PL/SQL, you can nest loops to multiple levels. You can nest FOR, WHILE, and basic loops within one another. The nested loop does not have to be the same type as the outer loop.

Example:

```
BEGIN
  FOR v_outerloop in 1..3 LOOP
    FOR v_innerloop in REVERSE 1..5 LOOP
      DBMS_OUTPUT.PUT_LINE('Outer loop is: ' || v_outerloop ||
                           'and inner loop is: ' || v_innerloop);
    END LOOP;
  END LOOP;
END;
```

 **Nested Loops continued**

This example contains EXIT conditions in nested basic loops.

```
DECLARE
  v_outer_done   CHAR(3) := 'NO';
  v_inner_done   CHAR(3) := 'NO';
BEGIN
  LOOP           -- outer loop
    ...
    LOOP       -- inner loop
      ...
      ...      -- step A
      EXIT WHEN v_inner_done = 'YES';
      ...
    END LOOP;
    ...
    EXIT WHEN v_outer_done = 'YES';
    ...
  END LOOP;
END;
```

What if we want to exit from the outer loop at step A?



Nested Loops continued

Use labels to distinguish between the loops:

```
DECLARE
...
BEGIN
  <<outer_loop>>
  LOOP          -- outer loop
    ...
    <<inner_loop>>
    LOOP        -- inner loop
      EXIT outer_loop when ... -- Exits both loops
      EXIT WHEN inner_done = 'YES';
      ...
    END LOOP inner_loop;
    ...
    EXIT WHEN outer_done = 'YES';
    ...
  END LOOP outer_loop;
END;
```



Guidelines for Loop Labels

Loop label names follow the same rules as other identifiers. A label is placed before a statement, either on the same line or on a separate line. In `FOR` or `WHILE` loops, place the label before `FOR` or `WHILE` within label delimiters (`<<label>>`). If the loop is labeled, the label name can optionally be included after the `END LOOP` statement for clarity.

Introduction to Explicit Cursors



Context Areas and Cursors

The Oracle server allocates a private memory area called a context area to store the data processed by a SQL statement.

Every context area (and therefore every SQL statement) has a cursor associated with it. You can think of a cursor either as a label for the context area, .

DML statements and single-row `SELECT` statements use an implicit cursor called SQL.

To `SELECT` more than one row, you must declare and use an explicit cursor.

 **Explicit Cursors continued**

With an explicit cursor, you can retrieve multiple rows from a database table, have a pointer to each row that is retrieved, and work on the rows one at a time.

The following are some reasons to use an explicit cursor:

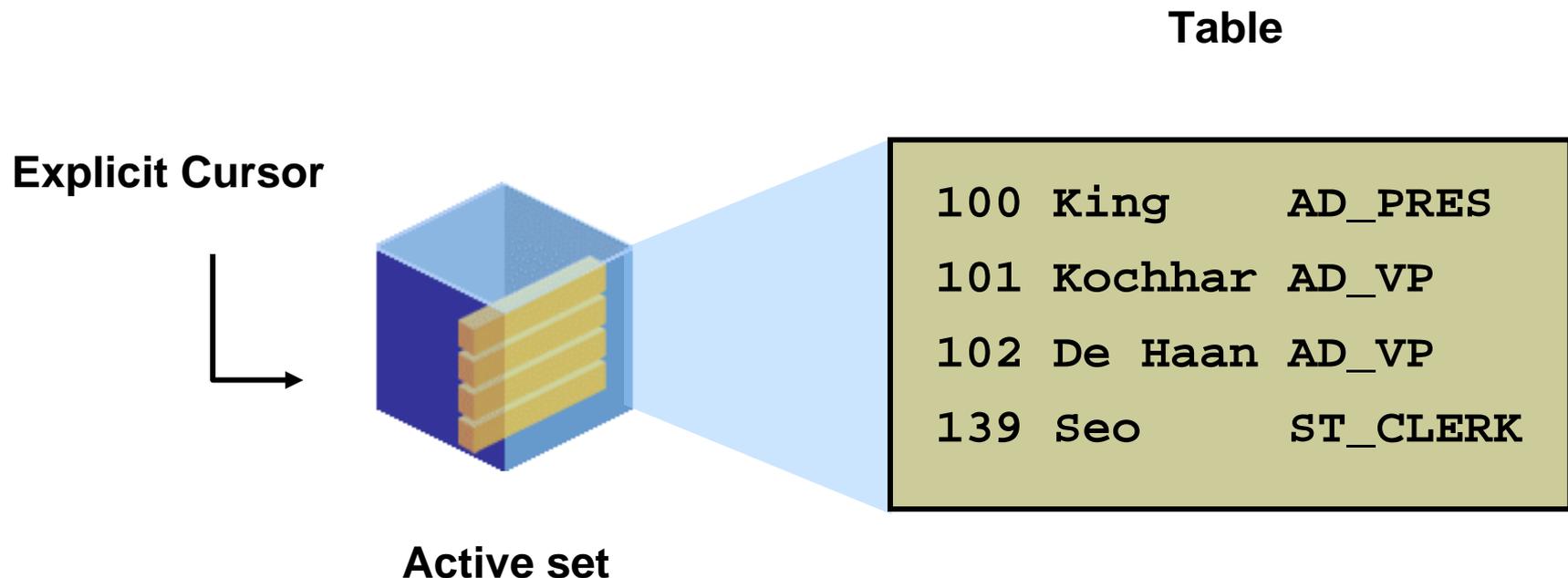
- It is the only way in PL/SQL to retrieve more than one row from a table.
- Each row is fetched by a separate program statement, giving the programmer more control over the processing of the rows.

 **Example:**

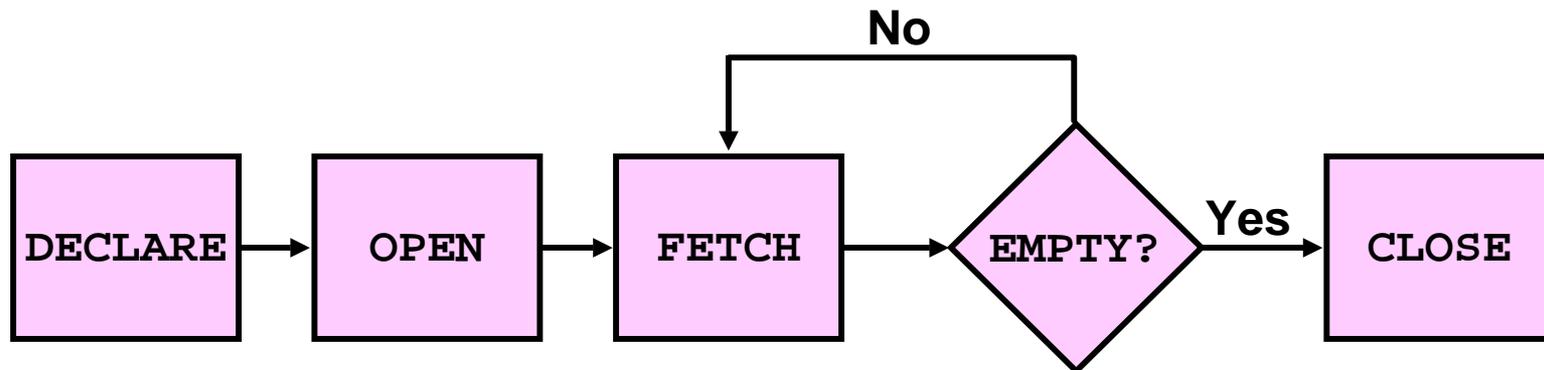
```
DECLARE
  CURSOR wf_holiday_cursor IS
  SELECT country_name, national_holiday_date
  FROM wf_countries WHERE region_id=30;
  v_country_name wf_countries.country_name%TYPE;
  v_holiday wf_countries.national_holiday_date%TYPE;
BEGIN
  OPEN wf_holiday_cursor;
  LOOP
    FETCH wf_holiday_cursor INTO v_country_name, v_holiday;
    DBMS_OUTPUT.PUT_LINE(v_country_name || ' ' || v_holiday);
    -- perform action on data here
    EXIT when wf_holiday_cursor%NOTFOUND;
  END LOOP;
  CLOSE wf_holiday_cursor;
END;
```

Explicit Cursor Operations

The set of rows returned by a multiple-row query is called the active set. Its size is the number of rows that meet your search criteria. The following illustration shows how an explicit cursor “points” to the current row in the active set. This enables your program to process the rows one at a time.



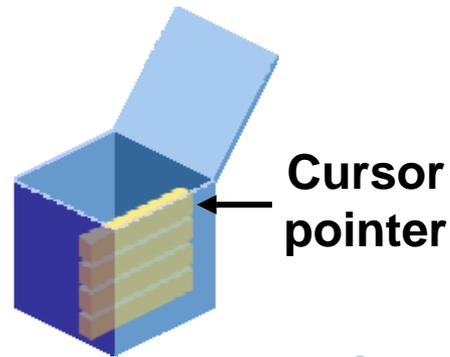
Controlling Explicit Cursors



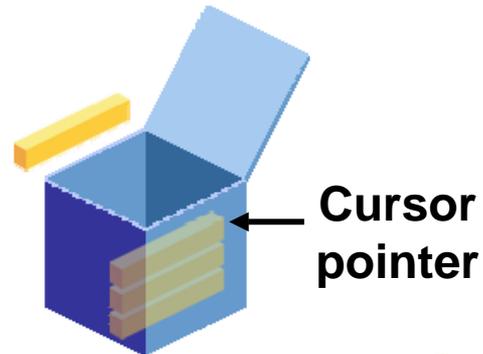
- Name an active set
- Fill the active set with data
- Retrieve the current row into variables
- Test for existing rows.
- Return to `FETCH` if rows are found.
- Release the active set.

Controlling Explicit Cursors continued

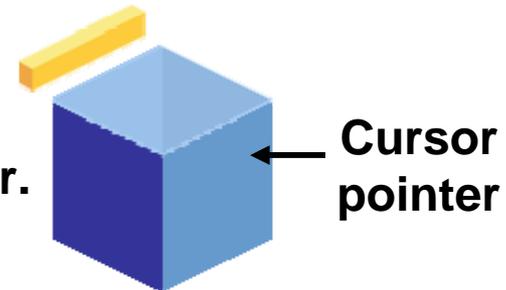
1 Open the cursor.



2 Fetch each row, one at a time.



3 Close the cursor.





Declaring the Cursor

The active set of a cursor is defined by the `SELECT` statement in the cursor declaration.

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

```
DECLARE
  CURSOR dept_emp_cursor IS
    SELECT department_name, COUNT(*) AS how_many
    FROM departments d, employees e
    WHERE d.department_id = e.department_id
    GROUP BY d.department_name
    HAVING COUNT(*) > 1;
...
```

The second example shows that a cursor declaration can include joins, group functions and subqueries. It can also include an `ORDER BY` clause, although that is not shown here.

Guidelines for Declaring the Cursor

- Do not include the `INTO` clause in the cursor declaration because it appears later in the `FETCH` statement.
- If processing rows in a specific sequence is required, then use the `ORDER BY` clause in the query.
- The cursor can be any valid `SELECT` statement, including joins, subqueries, and so on.



Opening the Cursor

The `OPEN` statement executes the query associated with the cursor, identifies the active set, and positions the cursor pointer to the first row. The `OPEN` statement is included in the executable section of the PL/SQL block.

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



Opening the Cursor continued

- OPEN is an executable statement that performs the following operations:
 1. Dynamically allocates memory for a context area
 2. Executes the `SELECT` statement in the cursor declaration, returning the results into the active set (fills the box with data)
 3. Positions the pointer to the first row in the active set.



Fetching Data from the Cursor

The `FETCH` statement retrieves the rows from the cursor one at a time. After each fetch, the cursor advances to the next row in the active set. You use the `%NOTFOUND` attribute to check whether the entire active set has been retrieved.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name FROM employees
       WHERE department_id =50;
  v_empno employees.employee_id%TYPE;
  v_lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_empno, v_lname;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno || ' ' || v_lname);
  END LOOP; ...
END;
```

```
124 Mourgos
141 Rajs
142 Davies
143 Matos
144 Vargas

Statement processed.
```

 **Guidelines for Fetching Data from the Cursor**

- Include the same number of variables in the `INTO` clause of the `FETCH` statement as columns in the `SELECT` statement, and be sure that the data types are compatible.
- Match each variable to correspond to the columns positionally.
- Test to see whether the cursor contains rows. If a fetch acquires no values, then there are no rows left to process in the active set and no error is recorded.



Closing the Cursor

The `CLOSE` statement disables the cursor, releases the context area, and undefines the active set. Close the cursor after completing the processing of the `FETCH` statement. You can reopen the cursor if required.

```
...  
  LOOP  
    FETCH emp_cursor INTO v_empno, v_lname;  
    EXIT WHEN emp_cursor%NOTFOUND;  
    DBMS_OUTPUT.PUT_LINE( v_empno || ' ' || v_lname);  
  END LOOP;  
  CLOSE emp_cursor;  
END;
```



Guidelines for Closing the Cursor

- A cursor can be reopened only if it is closed. If you attempt to fetch data from a cursor after it has been closed, then an `INVALID_CURSOR` exception will be raised.
- If we later reopen the cursor, the associated `SELECT` statement is re-executed to re-populate the context area with the most recent data from the database.

Using Explicit Cursor Attributes



Cursors and Records

Notice the difference between the two blocks of code. The code on the left uses hard-coded variables for each column selected in the cursor. The code on the right uses a `RECORD` for each column selected in the cursor. Records are convenient for processing the rows of the active set because you can simply fetch into the record.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name
       FROM employees
       WHERE department_id =30;
  v_emp_id      NUMBER;
  v_last_name   VARCHAR2(25);
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor
      INTO v_emp_id, v_last_name;
    ...
```

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name
       FROM employees
       WHERE department_id =30;
  v_emp_record  emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor
      INTO v_emp_record;
    ...
```



Explicit Cursor Attributes

These attributes are identical to those used with an implicit cursor. When appended to the cursor variable name, these attributes return useful information about the execution of a cursor manipulation statement.

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



%ISOPEN Attribute

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

%ISOPEN returns the status of the cursor: TRUE if open and FALSE if not.

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

 **%ROWCOUNT and %NOTFOUND Attributes**

Usually the %ROWCOUNT and %NOTFOUND attributes are used in a loop to determine when to exit the loop.

Use the %ROWCOUNT cursor attribute for the following:

- To process an exact number of rows
- To count the number of rows fetched so far in a loop and/or determine when to exit the loop

Use the %NOTFOUND cursor attribute for the following:

- To determine whether the query found any rows matching your criteria
- To determine when to exit the loop

 **Example of %ROWCOUNT and %NOTFOUND**

This example shows how %ROWCOUNT and %NOTFOUND attributes can be used for exit conditions in a loop.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name FROM employees;
  v_emp_record  emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_emp_record;
    EXIT WHEN emp_cursor%ROWCOUNT > 10 OR emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_emp_record.employee_id
      || ' ' || v_emp_record.last_name);
  END LOOP;
  CLOSE emp_cursor;
END ;
```

Explicit Cursor Attributes in SQL Statements

You cannot use an explicit cursor attribute directly in a SQL statement. The following code will return an error:

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, salary FROM employees
    ORDER BY SALARY DESC;
  v_emp_record  emp_cursor%ROWTYPE;
  v_count      NUMBER;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    INSERT INTO top_paid_emps
      (employee_id, rank, salary)
    VALUES
      (v_emp_record.employee_id, emp_cursor%ROWCOUNT,
      v_emp_record.salary);
    ...
  
```

Cursor FOR Loops



Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. The cursor is opened, a row is fetched once for each iteration in the loop, the loop is terminated automatically when the last row is processed, and the cursor is closed automatically.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name FROM employees
    WHERE department_id =50;
BEGIN
  FOR v_emp_record IN emp_cursor
  LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_record.employee_id
                          || ' ' || v_emp_record.last_name);
  END LOOP;
END;
```



Cursor FOR Loops continued

Compare this code to the previous slide. The two forms of the code are logically identical to each other and produce exactly the same results.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id,last_name
    FROM employees
    WHERE department_id =30;
  v_emp_record  emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(...);
  END LOOP;
  CLOSE emp_cursor;
END;
```



Guidelines for Cursor FOR Loops

- Do not declare the record that controls the loop because it is declared implicitly.
- The scope of the implicit record is restricted to the loop, so you cannot reference the record outside the loop.
- You can access fetched data by *record_name.column_name*.



Testing Cursor Attributes

We can still test cursor attributes such as %ROWCOUNT.

Example:

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name
    FROM employees
    WHERE department_id =50;
BEGIN
  FOR v_emp_record IN emp_cursor
  LOOP
    EXIT WHEN emp_cursor%ROWCOUNT > 5;
    DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                          || ' ' || v_emp_record.last_name);
  END LOOP;
END;
```



Cursor FOR Loops Using Subqueries

We don't have to declare the cursor at all! Instead, we can specify the `SELECT` on which the cursor is based directly in the `FOR` loop.

The advantage of this is that all the cursor definition is contained in a single `FOR ...` statement. This makes later changes to the code much easier and quicker.

```
BEGIN
  FOR v_emp_record IN (SELECT employee_id, last_name
                       FROM employees WHERE department_id =50)
  LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_record.employee_id
                          || ' ' || v_emp_record.last_name);
  END LOOP;
END;
```

Cursors with Parameters



Cursors with Parameters

You can pass parameters to a cursor when the cursor is opened. This means that you can open and close an explicit cursor several times in a block, or in different executions of the same block, returning a different active set on each occasion.

```
DECLARE
  CURSOR c_emp (p_dept_id NUMBER) IS
    SELECT first_name, last_name
      FROM employees
     WHERE department_id = p_dept_id;
  v_emp_record c_emp%ROWTYPE;
BEGIN
  OPEN c_emp (10);
  LOOP
    FETCH c_emp INTO v_emp_record;
    EXIT WHEN c_emp%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_emp_record.first_name || ' '
                        || v_emp_record.last_name);
  END LOOP;
  CLOSE c_emp;
END;
```

Change to whichever
department is required.



Defining Cursors with Parameters

Each parameter named in the cursor declaration must have a corresponding value in the `OPEN` statement. Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are used in the `WHERE` clause of the cursor `SELECT` statement.



Examples:

```
DECLARE
  CURSOR emp_cursor (p_deptno NUMBER) IS
    SELECT employee_id, last_name
      FROM employees
     WHERE department_id = p_deptno;
BEGIN
  OPEN emp_cursor (10);
  ...
  CLOSE emp_cursor;
  OPEN emp_cursor (20);
  ...
```

Open the cursor and return different active sets.

```
DECLARE
  CURSOR emp_cursor (p_deptno NUMBER) IS
    SELECT employee_id, last_name
      FROM employees
     WHERE department_id = p_deptno;
BEGIN
  FOR v_emp_record IN emp_cursor(10) LOOP
    ...
  END LOOP;
END;
```

 **Example:**

```
DECLARE
  v_deptid      employees.department_id%TYPE;
  CURSOR empcur (p_deptid NUMBER) IS
    SELECT employee_id, salary
           FROM my_employees
           WHERE department_id = p_deptid;
  v_emp_rec empcur%ROWTYPE;
BEGIN
  SELECT MAX(department_id) INTO v_deptid
     FROM my_employees;
  OPEN empcur(v_deptid);
  LOOP
    FETCH empcur INTO v_emp_rec;
    EXIT WHEN empcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.employee_id
                        || ' ' || v_emp_rec.salary);
  END LOOP;
  CLOSE empcur;
END;
```



Cursors with Multiple Parameters

In the following example, a cursor is declared and is called with two parameters:

```
DECLARE
  CURSOR    emp_cursor2 (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
  FOR v_emp_record IN emp_cursor2(10, 'Sales') LOOP
    ... i
  END LOOP;
END;
```

Using Cursors for Update

Declaring a Cursor with the FOR UPDATE Clause

When we declare a cursor `FOR UPDATE`, each row is locked as we `FETCH` it. This prevents other users from modifying the rows while our cursor is open. This does not prevent other users from reading the rows.

```
CURSOR cursor_name IS SELECT ... FROM ...  
FOR UPDATE [OF column_reference][NOWAIT | WAIT n];
```

- In the syntax:
 - *column_reference* Is a column in the table against which the query is performed (A list of columns may also be used.)
 - `NOWAIT` Returns an Oracle server error if any of the rows are locked by another session. If you omit the `NOWAIT` keyword, then the Oracle server waits indefinitely until all the rows are available.
 - `WAIT n` Specify the number of seconds *n* to wait and check whether the rows are unlocked. If the rows are still locked then an error is returned.



FOR UPDATE OF column-name

If the cursor is based on a join of two tables, we may want to lock the rows of one table but not the other. To do this, we specify any column of the one table we want to lock.

Example:

```
DECLARE
  CURSOR emp_cursor IS
    SELECT e.employee_id, d.department_name
       FROM employees e, departments d
       WHERE e.department_id = d.department_id
          AND department_id = 80 FOR UPDATE OF salary;
  ...
```



WHERE CURRENT OF Clause

The WHERE CURRENT OF clause in an UPDATE or DELETE statement is used in conjunction with the FOR UPDATE clause to refer to the current row in an explicit cursor.

```
WHERE CURRENT OF cursor-name ;
```

cursor Is the name of a declared cursor (The cursor must have been declared with the **FOR UPDATE** clause.)

This enables you to apply updates and/or deletes to the most recently **FETChED** row.



Example:

NOWAIT, FOR UPDATE, and WHERE CURRENT OF Clauses

```
DECLARE
  v_deptid      employees.department_id%TYPE;
  CURSOR empcur (p_deptid NUMBER) IS
    SELECT employee_id, salary
       FROM my_employees
       WHERE salary <= 20000
          AND department_id = p_deptid
          FOR UPDATE OF salary NOWAIT;
  v_emp_rec empcur%ROWTYPE;
BEGIN
  SELECT max(department_id) INTO v_deptid
     FROM my_employees;
  OPEN empcur(v_deptid);
  LOOP
    FETCH empcur INTO v_emp_rec;
    EXIT WHEN empcur%NOTFOUND;
    UPDATE my_employees
       SET salary = v_emp_rec.salary*1.1
       WHERE CURRENT OF empcur;
  END LOOP;
  CLOSE empcur;
  COMMIT;
END;
```

Using Multiple Cursors

 **Sample Problem 1**

We need to produce a report which lists each department as a sub-heading, immediately followed by a listing of the employees in that department ... followed by the next department, and so on.

Notice the second cursor contains a parameter so that each time it is opened it only contains employees of the referenced department.



Sample Problem 1 Cont

```
DECLARE
  CURSOR c_dept IS SELECT department_id,department_name
    FROM departments ORDER BY department_name;
  CURSOR c_emp (p_deptid NUMBER) IS SELECT first_name, last_name
    FROM employees
    WHERE department_id = p_deptid ORDER BY last_name;
  v_deptrec    c_dept%ROWTYPE;
  v_emprec     c_emp%ROWTYPE;
BEGIN
  OPEN c_dept;
  LOOP
    FETCH c_dept INTO v_deptrec;
    EXIT WHEN c_dept%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_deptrec.department_name);
    OPEN c_emp (v_deptrec.department_id);
    LOOP
      FETCH c_emp INTO v_emprec;
      EXIT WHEN c_emp%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_emprec.last_name || ' ' ||
                           v_emprec.first_name);
    END LOOP;
    CLOSE c_emp;
  END LOOP;
  CLOSE c_dept;
END;
```

 **Sample Problem 2**

We need to produce a report which lists each location in which our departments are situated, followed by the departments in that location.



Sample Problem 2 Cont

```
DECLARE
  CURSOR c_loc IS SELECT * FROM locations;
  CURSOR c_dept (p_locid NUMBER) IS
    SELECT * FROM departments WHERE location_id = p_locid;
  v_locrec    c_loc%ROWTYPE;
  v_deptrec   c_dept%ROWTYPE;
BEGIN
  OPEN c_loc;
  LOOP
    FETCH c_loc INTO v_locrec;
    EXIT WHEN c_loc%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_locrec.city);
    OPEN c_dept (v_locrec.location_id);
    LOOP
      FETCH c_dept INTO v_deptrec;
      EXIT WHEN c_dept%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_deptrec.department_name);
    END LOOP;
    CLOSE c_dept;
  END LOOP;
  CLOSE c_loc;
END;
```



Using FOR Loops with Multiple Cursors

We can use FOR loops (and other cursor techniques such as FOR UPDATE) with multiple cursors, just as we can with single cursors.

```
DECLARE
  CURSOR c_loc IS SELECT * FROM locations;
  CURSOR c_dept (p_locid NUMBER) IS
    SELECT * FROM departments WHERE location_id = p_locid;
BEGIN
  FOR v_locrec IN c_loc
  LOOP
    DBMS_OUTPUT.PUT_LINE(v_locrec.city);
    FOR v_deptrec IN c_dept (v_locrec.location_id)
    LOOP
      DBMS_OUTPUT.PUT_LINE(v_deptrec.department_name);
    END LOOP;
  END LOOP;
END;
```